



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**CONFIGURATION TOOL PROTOTYPE FOR THE
TRUSTED COMPUTING EXEMPLAR PROJECT**

by

Terrence M. Welliver

December 2009

Thesis Advisor:
Second Reader:

Cynthia E. Irvine
Paul C. Clark

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 2009	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Configuration Tool for the Trusted Computing Exemplar Project		5. FUNDING NUMBERS	
6. AUTHOR(S) Terrence M. Welliver			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>The creation of a configuration vector file used to initialize the Least Privilege Separation Kernel (LPSK) of the Trusted Computing Exemplar (TCX) project to an initial secure state is currently a manual process that is tedious and error prone. A software application that removes many of the complexities of creating a valid configuration vector file is needed.</p> <p>This thesis describes the first steps taken to design and implement a graphical user interface (GUI) configuration vector tool that enables a user to easily create valid configuration vector files (both human-readable and binary). The tool allows a user to focus on the meaning of the configuration vector rather than on the syntactic details of the file.</p> <p>A prototype of the configuration vector tool was successfully designed, implemented, and tested in this thesis. The prototype provides the first functional GUI software application that creates configuration vector files. The logical design of the toll will permit further extensions to be readily incorporated.</p>			
14. SUBJECT TERMS trusted computing exemplar, least privilege separation kernel, graphical user interface, wxpython, java, configuration vector, lpsk, configuration vector tool, tcx, gui, skpp			15. NUMBER OF PAGES 119
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**CONFIGURATION TOOL PROTOTYPE FOR THE TRUSTED COMPUTING
EXEMPLAR PROJECT**

Terrence M. Welliver
Civilian, Naval Postgraduate School
B.S., United States Air Force Academy, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
December 2009**

Author: Terrence M. Welliver

Approved by: Cynthia E. Irvine
Thesis Advisor

Paul C. Clark
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The creation of a configuration vector file used to initialize the Least Privilege Separation Kernel (LPSK) of the Trusted Computing Exemplar (TCX) project to an initial secure state is currently a manual process that is tedious and error prone. A software application that removes many of the complexities of creating a valid configuration vector file is needed.

This thesis describes the first steps taken to design and implement a graphical user interface (GUI) configuration vector tool that enables a user to easily create valid configuration vector files (both human-readable and binary). The tool allows a user to focus on the meaning of the configuration vector rather than on the syntactic details of the file.

A prototype of the configuration vector tool was successfully designed, implemented, and tested in this thesis. The prototype provides the first functional GUI software application that creates configuration vector files. The logical design of the tool will permit further extensions to be readily incorporated.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	MOTIVATION	2
B.	PURPOSE OF STUDY.....	3
C.	THESIS ORGANIZATION.....	3
II.	BACKGROUND	5
A.	TRUSTED COMPUTING EXEMPLAR PROJECT.....	5
1.	Trusted Computing Overview	6
2.	Separation Kernel	8
3.	The TCX LPSK.....	10
4.	Configuration Vector Tool.....	11
B.	GRAPHICAL USER INTERFACES.....	12
1.	Model-View-Controller Paradigm	13
2.	Java and MVC.....	15
3.	GUI Terminology	16
C.	SUMMARY	18
III.	DESIGN	19
A.	OVERVIEW OF DEVELOPMENT TOOLS	19
1.	Tool Selection Process.....	19
2.	Java Swing and NetBeans	23
B.	CONFIGURATION VECTOR FORMAT.....	24
1.	Basics.....	24
2.	Structure Overview.....	25
3.	MVC Model Component	29
C.	THE PRIMARY GRAPHICAL DESIGN ELEMENT	29
D.	DESIGN REFINEMENT	30
E.	CONFIGURATION VECTOR TOOL REQUIREMENTS	38
1.	Basic Requirements	38
2.	Detailed Requirements	41
3.	Error Message Requirements	45
F.	CONFIGURATION VECTOR TOOL FEATURE SET	47
1.	Minimum Feature Set.....	47
2.	Features Users Expect	48
3.	Graphical Interface Standards	49
G.	CONCEPTUAL DESIGN OF THE CONFIGURATION VECTOR TOOL	50
H.	SUMMARY	56
IV.	IMPLEMENTATION AND TESTING	57
A.	JAVA CLASS FILES	57
B.	PRIMARY GUI CLASS.....	59
C.	PROTOTYPE.....	62
1.	Screenshots	62

2.	Concept of Operation	67
D.	TESTING.....	69
1.	Phase I: Error Checking	70
2.	Phase II: Input/Output	76
3.	Test Summary	80
E.	SUMMARY	81
V.	RESULTS	83
A.	PROBLEMS ENCOUNTERED	83
1.	wxPython	83
2.	NetBeans Tables	84
B.	INCOMPLETE FEATURES	84
C.	FUTURE WORK	86
1.	Interface	86
2.	Additional Features	87
3.	Refinements	88
4.	Documentation	88
D.	CONCLUSION	89
	APPENDIX.....	91
	LIST OF REFERENCES	97
	INITIAL DISTRIBUTION LIST	99

LIST OF FIGURES

Figure 1.	An example of the allocation of subjects and resources along with the information flow in a separation kernel [4]	9
Figure 2.	Illustrates the lifecycle of the configuration vector from initial creation to the initialization of the LPSK to a secure state.....	11
Figure 3.	MVC interaction diagram [6]	14
Figure 4.	Java Swing MVC based architecture diagram [7]	16
Figure 5.	GUI elements referenced in this thesis	17
Figure 6.	Configuration vector excerpt from <code>lpsk.h</code>	26
Figure 7.	Configuration vector structure diagram based on the <code>lpsk.h</code>	28
Figure 8.	Illustrates the transformation of the <code>vector_struct</code> attributes to table column headers	30
Figure 9.	Illustration depicting the incorporation of the tables and <code>vector_struct</code> attributes into a main application window	32
Figure 10.	Illustration of the nesting child tables within parent tables	33
Figure 11.	A refinement of the initial window design showing the addition of the partition-to-partition table.....	34
Figure 12.	A refinement of the process and subject tables. The two tables are merged into one window	35
Figure 13.	All permissions columns within each table is represented by a common interface window	36
Figure 14.	Final table column headers as defined by the <code>lpsk.h</code> file	37
Figure 15.	A complete design concept of the <code>vector_struct</code>	38
Figure 16.	Configuration vector tool state transition diagram	40
Figure 17.	Conceptual sketch of the partition table	51
Figure 18.	Conceptual sketch of the datafile table.....	52
Figure 19.	Conceptual sketch of the memory table.....	52
Figure 20.	Conceptual sketch of the event counts table.....	53
Figure 21.	Conceptual sketch of the sequencers table	53
Figure 22.	Conceptual sketch of the partition-to-partition table	54
Figure 23.	Conceptual sketch of the subject resource table	54
Figure 24.	Conceptual sketch of the processes window and subjects table	55
Figure 25.	Conceptual sketch of the permissions window and table	55
Figure 26.	Partition table view of the application	62
Figure 27.	Datafile table view of the application.....	63
Figure 28.	Memory table view of the application	63
Figure 29.	Eventcounts table view of the application	64
Figure 30.	Sequencers table view of the application.....	64
Figure 31.	Partition-to-partition table view of the application.....	65
Figure 32.	Subject-resource permissions table view of the application	65
Figure 33.	Process and subject window of the application	66
Figure 34.	Permissions window and associated table view of the application	66
Figure 35.	The vector attribute panel of the main window	67
Figure 36.	View of row zero of the partition table.....	68

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Candidate development language criteria	20
Table 2.	Candidate development languages comparison	21
Table 3.	The possible output file type given a specific input file type	40
Table 4.	Constant values defined by the <code>lpsk.h</code> file	41
Table 5.	Mapping of permissions from the human-readable format to the binary format.....	42
Table 6.	Upper and lower bounds for all objects contained within every struct of the <code>vector_struct</code>	44
Table 7.	Dependency relationships of configuration vector fields	45
Table 8.	Minimum configuration vector tool features	48
Table 9.	Java class files of the configuration vector tool.....	59
Table 10.	List of the basic commands of the configuration vector tool.....	61
Table 11.	Vector attributes panel restrictions	70
Table 12.	Partition table restrictions	71
Table 13.	Datafile table restrictions	71
Table 14.	Memory table restrictions	72
Table 15.	Eventcounts table restrictions	72
Table 16.	Sequencer table restrictions	72
Table 17.	Partition-to-partition table restrictions	72
Table 18.	Subject-resource table restrictions	73
Table 19.	Process window restrictions.....	73
Table 20.	Permissions window restrictions.....	73
Table 21.	Special tests table for starred entries.....	76
Table 22.	Commands used to verify the CVDump tool.....	77
Table 23.	Verification of CVDump command line tool	77
Table 24.	Test results for creating a new configuration vector.....	78
Table 25.	Test results for opening a configuration vector	79
Table 26.	Test results for saving a configuration vector.....	80
Table 27.	Test results for exporting a configuration vector.....	80

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
AWT	Abstract Window Toolkit
CC	Common Criteria
EAL	Evaluation Assurance Level
GUI	Graphical User Interface
IAD	Information Assurance Directorate
IDE	Integrated Development Environment
IO	Input/Output
IT	Information Technology
JDK	Java Development Kit
LPSK	Least Privilege Separation Kernel
MVC	Model-View-Controller
NSA	National Security Agency
PIFP	Partitioned Information Flow Policy
SKPP	Separation Kernel Protection Profile
SMDN	Sun Microsystems Developer Network
SPT	Special Tests
TCX	Trusted Computing Exemplar
TPA	Trusted Path Application
UI	User-Interface

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Cynthia Irvine, for her outstanding support and guidance throughout the development of this thesis. I would also like to thank Paul Clark for his support and insights as a second reader. A special thanks to Valerie Linhoff for supporting me during this whole process. In addition, I would like to thank Lt Col Joel Young for pointing me in the right direction during the programming phase of this thesis. Finally, I would like to thank my wife, Beth, for her support and patience throughout the thesis process.

This material is based upon work supported by the National Science Foundation, under grant No. DUE-0414102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author, and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Graphical user interfaces (GUI) have changed the way people interact with computers. Unlike command line interfaces, GUIs do not require a user to learn complicated sequences of words and symbols to complete an operation. Thus, the learning curve for a GUI application is significantly less steep than for a command line interface. A GUI also abstracts away the sometimes-complicated syntax necessary for completing some task.

For instance, suppose a corporation wishes to collect specific information from several different users and analyze it. The corporation could request the required information from each user by individually asking each user to provide a, b, and c. The corporation could go even one step further and request the information in a specific format. While this approach may work, it is not very efficient. Users are prone to make errors, especially when it comes to correctly formatting data. Most likely, the corporation will receive the information it desires in a variety of formats. If the corporation receives the requested data in the wrong format, someone would need to transpose the data into the correct format. This increases the cost of obtaining and analyzing the required data. Thus, a different approach is necessary.

Instead, the corporation could create a GUI form and require users to enter data in the form. The form graphically organizes the data for the user. The user will most likely find this method of data collection much easier than a simple request for data from the corporation. In addition, the underlying code of the form will automatically ensure that the data is in the correct format. This allows the corporation to collect the desired data from the users more effectively and efficiently. Thus, the corporation can spend the majority of time analyzing the data.

The example above reflects goals similar to those presented in this thesis. In this case, the goal is to easily create configuration files (for a particular kind of system) that contain no syntax errors. The configuration file for the system of interest is referred to as a configuration vector. This file is in a binary format and is used to initialize an

instantiation of the system to an initial secure state. The configuration vectors contain a substantial amount of data that must be entered by a user. The configuration vector is syntactically complex and requires the user to be very meticulous during its creation. In order to focus the user on the semantic meaning of the configuration vector, the creation of a GUI tool to assist in configuration vector definition was proposed. It is expected that such a tool would ensure that data entered by the user is exported in the correct format. The next section describes the motivation for this research followed by the purpose of the study.

A. MOTIVATION

The Trusted Computing Exemplar (TCX) project provides an example of how high assurance components are designed and built. One of the main components of the project is the Least Privilege Separation Kernel (LPSK). The LPSK controls the flow of information between resources by separating system resources into different subsets called *partitions*. The LPSK is initialized to an initial secure state through the use of a binary configuration vector file.

Without a GUI, a trusted user initially creates a human-readable configuration vector file. This human-readable configuration vector would then be converted into a binary file that the LPSK can consume. The process of creating a configuration vector file manually is likely to be a tedious, error-prone, and time-consuming process. This process can be improved upon. One approach to create configuration vector files more efficiently is to create a configuration vector definition tool that allows trusted users to enter data and review graphically.

The configuration vector tool is an application that provides a GUI to the trusted user to be used to create a valid configuration vector file. The configuration vector tool provides essential bounds checking in order to ensure that a syntactically correct binary configuration vector file is generated. This thesis describes the creation of such a tool.

In order to accommodate future changes to either the interface (view) or the data structure (model) of the application, standard design guidelines that separate view components from model components were followed. This enables the configuration vector tool to be easily modified to enhance usability or add new features.

B. PURPOSE OF STUDY

The objective of this research was to analyze the configuration vector currently used by the LPSK and to create an application that provides a GUI to the trusted user. This graphical interface provides an additional layer of abstraction to the trusted user, who can then spend more time thinking about correct configurations rather than worrying about syntax details. This tool should reduce the time it takes to create a configuration vector while also helping the user make fewer configuration mistakes.

C. THESIS ORGANIZATION

Chapter I contains a brief introduction of the work along with its motivation and organization. Chapter II provides the necessary background and foundation for understanding the purpose of this research, which includes a brief overview of trusted computing and separation kernels as well as an overview of graphical user interface design. Chapter III contains the specific steps taken to develop the first prototype of the configuration vector tool. The chapter begins with the language and development tool selection. Next, the configuration vector format is discussed in detail. The next portion describes the thought process behind the initial design and the refinement of that design including the requirements and features for the configuration vector tool. Chapter IV presents the implementation and testing of the configuration vector tool prototype. Chapter V presents the main problems encountered as well as future implementation and design ideas for the next version of the configuration vector tool.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter provides the necessary foundation for this research in two sections. The first section of the chapter discusses the Trusted Computing Exemplar (TCX) project by first examining trusted computing and then discussing separation kernels. The section then discusses the TCX implementation of a separation kernel. The section concludes with the introduction of the configuration vector tool, which initiates the TCX separation kernel to an initial secure state. The second section of the chapter discusses graphical user interfaces (GUI). The section begins with a paradigm discussion followed by the Java implementation of the paradigm. Finally, the section gives a brief overview of the GUI terminology relevant to this thesis.

A. TRUSTED COMPUTING EXEMPLAR PROJECT

The TCX project is intended to be a worked example of how high assurance components are designed and built. Not only will the TCX project provide components for use in current architectures, it is intended to ensure that knowledge and technologies associated with high assurance system development are available for future generations of developers and researchers [1]. The development methodologies and technologies from the TCX will be openly available and will fill a void in knowledge and research left by limited or closed development of trusted systems by the U.S. Government and commercial entities, respectively. The TCX objectives, listed below, highlight the project's overall goal of creating development methodologies and technologies.

- Creation of a prototype framework for rapid high assurance system development;
- Development of a reference implementation trusted computing component;
- Evaluation of the component for high assurance; and
- Open dissemination of deliverables related to the first three activities [1].

The next three sections provide an overview of trusted computing, a foundation for the TCX platform (a separation kernel), and a description of the TCX separation kernel implementation.

1. Trusted Computing Overview

Trusted computing is an active area of research in computer science. Blindly trusting a system by examining marketing documents and user manuals, and listening to vendors describe a product's security controls does not provide customers with sufficient confidence that critical policies for the protection of information in information technology (IT) systems are enforced. Trusted computing assures customers that the computer will behave in accordance with a given security policy. Thus, trusted computing addresses security at the most fundamental level.

In order to better understand trusted computing, *trust* and *trustworthy* must be clearly defined. Imprecise definitions of these terms often lead to confusion or misinterpretation. In addition, it is important to understand what makes a system untrustworthy and how a system can be made more trustworthy. In the information security world, the term *trust* is sometimes used inconsistently; but, for the average user, trust simply means that the user believes the system is secure and the user can confide personal information in that system [2]. In order to understand the TCX project, a precise definition of trust is required.

trust: the degree to which the user or a component depends on the trustworthiness of another component. For example, component A trusts component B, or component B is trusted by component A. Trust and trustworthiness are assumed to be measured on the same scale [2].

trustworthy: the degree to which the security behavior of the component is demonstrably compliant with its stated functionality (*e.g.*, *trustworthy component*) [2].

From these definitions, a component is only trustworthy if and only if its functionality is exactly as described in its specification and the component has no other functionality. In other words, a highly trustworthy component will only do exactly what its specification states and nothing else [1]. Although this is conceptually simple, the implementation of such a component is difficult. Trusted systems enforce a given security policy through hardware and software.

Today's most popular operating systems have hundreds of thousands, or even millions of lines of code. Information security specialists and system designers are unable to demonstrate that every component in the system is trustworthy and therefore can be trusted. Thus, attempting to design highly trustworthy systems without understanding and meeting the requirements is futile. Systems that cannot be demonstrated to be trustworthy are considered untrustworthy. One approach to developing more trustworthy systems involves creating small (relative to other systems) and analyzable components. By creating such components, it is possible to create more trustworthy systems.

Trusted system development is a design and engineering approach to develop more trustworthy systems. Development processes need to address two different security threats: operational and developmental [1]. A frontal attack is an example of an operational threat. Frontal attacks are what the average person associates with computer security. These attacks can range from computer viruses and worms to Trojan horses to denial-of-service attacks. Frontal attacks exploit flaws in system code, configuration errors, and operator errors—threats from outside the system [1]. In order to address these operational issues, systems must have no exploitable flaws, constrain access to information, and isolate damage from malicious software execution [1].

A developmental threat can be much more dangerous. A system is susceptible to this threat during its development phase before the system becomes operational. This means that intentional exploits or malicious code inserted into the system during development could still potentially be in the system during the operational phase. An example of a developmental threat is subversion. Subversion is a malicious attempt to undermine system security policies and protections. Subversion manifests itself in the form of artifices (seemingly useless or unnoticed snippets of code) placed in the system anytime during the development lifecycle. A system containing artifices may have some or all installed security controls bypassed.

The assurance requirements of the Common Criteria (CC) [3] were created to address developmental threats. The CC defines assurance requirements for developing, implementing, evaluating, and maintaining trustworthy systems. Vendors who develop

systems to meet the standards specified in the CC can provide varying levels of assurance to customers that the specification, implementation, and evaluation of the product were conducted meticulously and in a standard manner. Unfortunately, it is not possible to provide 100% assurance that no artifices exist in a system. However, the CC includes evaluation assurance levels (EALs) that define a range of activities a vendor can employ when developing a system, from those that result in a very low assurance product (EAL1) to those that result in a very high assurance product (EAL7). Thus, insisting that products are evaluated against the appropriate CC EALs provides a higher level of confidence that assurance requirements are met than a product that has not complied with CC EALs.

Trusted computing is an attempt to systematically minimize operational and developmental threats. Because examples of high assurance development are not widely available, the TCX project attempts to solve the operational and developmental issues discussed above by creating an example of all elements of a high assurance system development. The next section provides context for the TCX Least Privilege Separation Kernel (LPSK).

2. Separation Kernel

In June 2007, the Information Assurance Directorate (IAD) of the National Security Agency (NSA) published the “*U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*” (SKPP) in order to define stringent requirements for high-assurance separation kernels [4]. Any separation kernel that conforms to the SKPP will provide a high degree of assurance that the system security policy is strictly enforced.

Separation kernels are different from typical security kernels that dynamically conduct all of the security functions in a system. Based on an input configuration, a separation kernel enforces an information flow policy by allocating the subjects and other exported resources of a system to partitions [4]. By doing so, a separation kernel can isolate subjects and other exported resources from one another and control the information flows (if any) between subjects and other resources. Subjects are the

The configuration vector specifies the partitioning of the system resources. In addition, the configuration vector specifies the Partitioned Information Flow Policy (PIFP), which is the allowed information flow between the partitions [4]. A configuration vector is translated from a human-readable form to a machine-readable format by the configuration function [4]. The policy can be expressed in terms of a partition-to-partition flow policy as well as a more granular subject-resource policy. Each configuration vector contains the appropriate information to initialize the system into a secure state. Only one binary configuration vector from the set of configuration vectors is selected to initialize the system. The next section describes the TCX implementation of a separation kernel that complies with the SKPP.

3. The TCX LPSK

The TCX project expands on the functional requirements outlined in the SKPP and implements a LPSK that is compliant with the SKPP. The LPSK provides hosted applications a high-degree of assurance that the PIFP is strictly enforced. The LPSK follows guidelines of the SKPP discussed in the previous section.

The PIFP for the LPSK is defined in the configuration data. Specifically, the configuration data is the result of setting the platform to an initial secure state from information contained in a binary configuration vector. Thus, a binary configuration vector contains the binary information that specifies the initial secure state of the LPSK. Figure 2 shows the steps necessary to initialize the LPSK to a secure state from the initial creation of the human-readable configuration vector to the creation of the binary configuration vector to the initialization of the LPSK to a secure state.

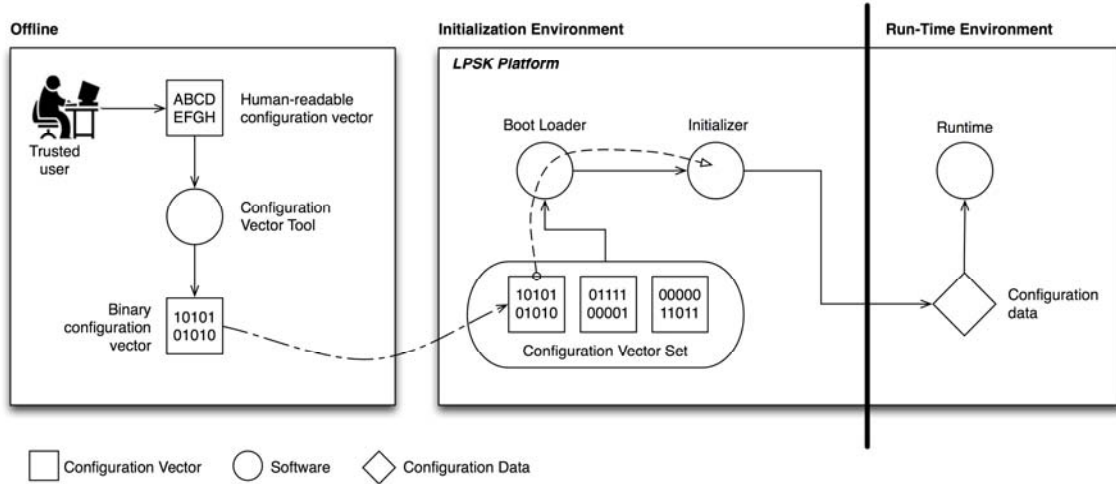


Figure 2. Illustrates the lifecycle of the configuration vector from initial creation to the initialization of the LPSK to a secure state

As shown in Figure 2, the initial configuration vector, which is human-readable, is created offline by a trusted user. The configuration vector is transformed from a human-readable format to a machine-readable binary format by the configuration vector tool. The resulting vector is transported to the LPSK platform. The LPSK Boot Loader is the first software to execute on the LPSK platform. The Boot Loader presents the configuration vector set to an authorized user. The configuration vector set is a collection of binary configuration vectors (only three are shown in Figure 2) that is presented to an authorized user during initialization. The authorized user must select the desired binary configuration vector from the configuration vector set. After the authorized user selects the desired configuration vector, the Boot Loader loads and starts the LPSK Initializer. The Initializer sets the LPSK platform to the initial secure state based on the data in the binary configuration vector. Once this is complete, the LPSK platform is able to enforce the loaded configuration in order to maintain a secure state. The next section discusses the first steps of this process—the conversion of the human-readable configuration vector into the binary configuration vector via the configuration vector tool application.

4. Configuration Vector Tool

The configuration vector tool is the primary focus of this research. As mentioned in the previous section, the configuration vector contains the pertinent information used

to initialize the LPSK platform to a secure state. The format of the configuration vector when loaded into the LPSK platform is binary. A binary format is neither easily created nor easily understood by users. Thus, a human-readable configuration vector is desirable for human interaction. However, in order to convert a human-readable configuration vector to a binary configuration vector, the human-readable configuration vector must adhere to a precise format. Only a precise human-readable format will allow a binary converter tool to faithfully convert the file to a binary format. This requires an authorized user to be very meticulous when creating the human-readable configuration vector. In general, users are not always proficient when it comes to completing these types of tasks.

To reduce the human errors that could occur when creating configuration vectors, this research aims to take the initial steps to construct an interface that allows a trusted user to create configuration vectors graphically. This graphical tool would allow the user to create a new configuration vector, or read in an existing binary configuration vector, and write out a binary configuration vector. This tool would also perform basic consistency checking before the configuration, created at the interface, is exported to a binary format. This ensures that every binary vector exported from the tool is in the correct format and has no syntactic errors. Eventually, the tool should be able to further check the created configuration for undesirable or unintended configurations.

Thus, this research intends to implement the configuration vector tool for the TCX project. The focus of the research is to create a user-friendly configuration vector tool that gathers the required data from a user and generates a correct binary configuration vector file. A user-friendly environment assumes the need to create a GUI application. The next section explores the graphical interface concepts relevant to the creation of the initial configuration vector tool.

B. GRAPHICAL USER INTERFACES

GUIs have changed the way people use computers. Instead of having to remember long sequences of strings, users can simply “point-and-click” their way to accomplish the majority of tasks on today’s systems. GUIs attempt to abstract away the complexities of dealing with the command line where a user must remember specific

strings in order to accomplish tasks. Although there are many different tools to assist users in constructing GUI applications, the underlying paradigm used to construct a well-designed GUI application has not changed much in the last few decades [5]. The Model-View-Controller (MVC) paradigm enables a developer to create a flexible and robust application [5]. The following section explains the concepts behind the paradigm and the next section discusses the Java implementation of the paradigm. The final part of this section outlines the basic terminology for discussing GUIs.

1. Model-View-Controller Paradigm

Most GUIs designed today follow, or attempt to follow, the MVC design paradigm. MVC is a simple and elegant approach to designing GUIs [5]. Smalltalk developer Trygve Reenskaug originally conceived the MVC concept in 1979 at the Xerox Palo Alto Research Center [5]. In 1979, GUI applications were rare and the concept of how to design one was virtually nonexistent. MVC divides modeling the external world, user input, and visual feedback into three distinct components: the model, the view, and the controller [5].

The *model* component is concerned with the data of the application, the access to that data, and the manipulation of that data. The model is essentially a software instantiation of the real-world process [6]. The model maintains the data and responds to requests to use the data.

The *view* component is concerned with managing and generating the GUI. The view specifies exactly how the information from the model is displayed on the given interface. It is important that the view presents the data in a consistent and uniform fashion in order to increase usability and reduce confusion. The view may query the model for data. However, the view cannot directly change the model data. Instead, the view sends events to the controller component.

The *controller* component listens for input from the view (via a mouse click, keyboard input, or an other event) and commands the model or view (or both) to perform a specific action [5]. The actions performed by the model are usually related to changing

the state of its data while the actions of the view usually prompt the view to change the state of its visual representation of data [6].

Although separated, each component must maintain contact with the other components in order function properly. Figure 3 shows the basic interaction between each aspect of the MVC. In Figure 3, the dotted lines represent events or notifications while the solid lines indicate specific method invocations. The solid line from the view to the model indicates a query of the model's data while the solid lines from the controller represent actions sent to the model or view.

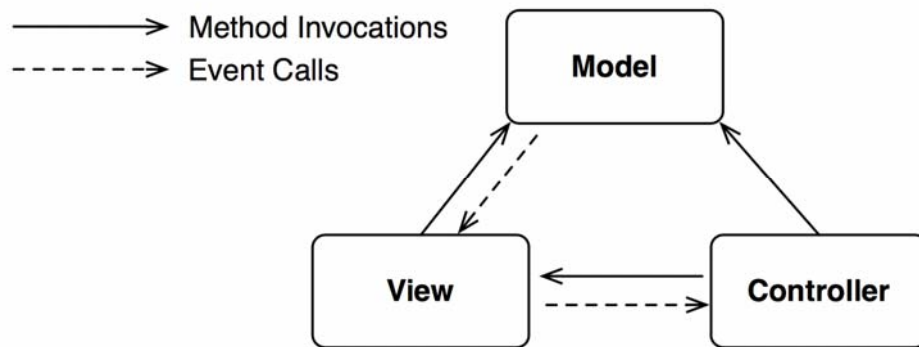


Figure 3. MVC interaction diagram [6]

Although the MVC paradigm is quite simple, its consequences are profound. An application that follows MVC is inherently very flexible. MVC design allows for the reuse of model components since the model and view are separate. This separation allows developers to create multiple views that access the same data. Since the model contains no complex GUI code, the model components are also much easier to maintain, debug, and test. The application has the ability to support different interfaces as well as different functionality by writing new view and controller code. A negative consequence to MVC is the increase of design complexity. Increased complexity introduces not only extra code in order to separate the model, view, and controller, but it also increases the time necessary to develop and implement an application. However, the negative consequences are minor relative to the advantages gained by following the MVC guidelines.

2. Java and MVC

Java uses the Swing architecture for its GUI development. The Swing architecture has its roots in MVC [7]. The design goals of the Swing project were to implement GUI components completely in Java to enable cross-platform compatibility, provide a single application programming interface (API) that supported multiple views, enable model-driven programming, ensure components behaved well in development tools (i.e., IDEs), and provide backward compatibility with the abstract window toolkit (AWT) APIs [7].

The developers of Swing realized that the MVC design was the most appropriate paradigm choice as it met the design goals stated above. However, while the MVC is conceptually the best structure, the developers quickly discovered that a complete split into the three components was impractical. A simple example using a Swing TextField component illustrates this problem. A TextField is used to display existing data and to change data. Since a TextField displays data, it must be a view component. However, it should also have the ability to change data in the model. Hence, it must be a controller component. This means that the TextField is sometimes a view component and sometimes a controller component. A TextField easily belongs to both categories.

The developers realized that a tight relationship existed between the view and controller components. Because of this relationship, writing a generic controller that had no knowledge of specific view items was very difficult. To solve this problem, the development team collapsed the view and controller components into a single user-interface (UI), referred to as a delegate—shown in Figure 4 [7]. Collapsing the view and controller components allows the components to efficiently communicate with one another. The delegate component communicates with separate model component as a single component.

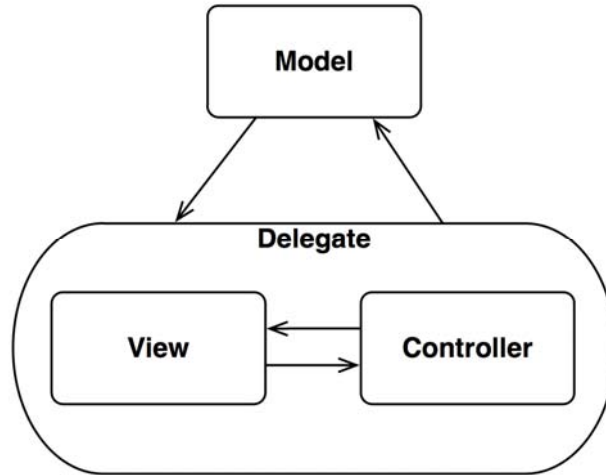


Figure 4. Java Swing MVC based architecture diagram [7]

Thus, Java Swing does not strictly follow the MVC paradigm. Instead, Swing is MVC based. This is usually referred to as a separable model architecture [7]. Although collapsing the view and controller into a single delegate may seem to present a problem to developers wishing to embrace Swing for development purposes; the two collapsed components (view and controller) of the delegate component may be treated somewhat independently [7]. In other words, even though the delegate actually contains both the view and controller simultaneously, the developer can access the view and controller functions of the delegate independently (in most cases).

3. GUI Terminology

This section briefly introduces terminology used to describe GUIs. Specifically, this section provides the basic terminology used throughout this thesis to describe the GUI elements of the configuration vector tool. The main element of a GUI is the *window*. A window is referred to as a *frame* in Java, but for the purposes of this thesis, a window (Java frame) will be referred to as a window. A window is the element that contains all other elements of a GUI (i.e., a window is what is seen on a display). Inside a window, many different types of elements may exist. For this research, the primary elements are panels, tables, labels, menus, buttons, combo boxes, spinners, and text fields. Panels are a way of grouping several elements together and are usually positioned

directly on the window. Tables are used to organize data in a tabular format and can be either read-only or editable. Labels are non-editable fields that provide the information as to what type of data is required for a specific field. Menus and buttons provide a way to execute events or actions for an application (such as opening or saving a file). Combo boxes and spinners help restrict the input a user enters into the application. The final element of importance to this thesis is a text field. A text field may be represented in a variety of ways and can be read-only or editable. Editable text fields allow users to enter data into the text field just as an editable table allows users to enter data into its cells. There are many other types of GUI elements that were not discussed in this section. Figure 5 shows the basic GUI elements referenced by this thesis. The elements discussed in this section are referenced throughout Chapter III and Chapter IV.

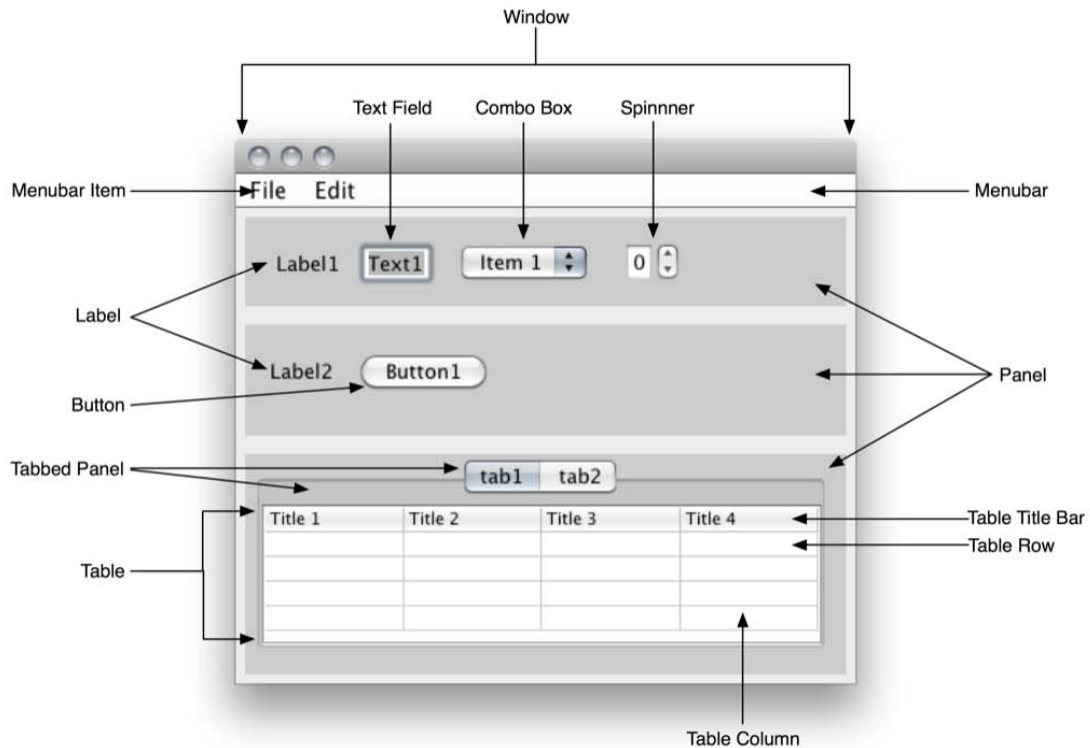


Figure 5. GUI elements referenced in this thesis

C. SUMMARY

This chapter presented a brief overview of the TCX objectives, a discussion of trusted computing, and then a discussion of the TCX LPSK and, in particular, of the configuration vector. The chapter then presented the MVC design paradigm for GUI development and the Java implementation of the MVC paradigm: Swing. The chapter ended by introducing basic GUI terminology. The next chapter presents the design considerations for creating the GUI configuration vector tool.

III. DESIGN

LPSK developers and potential users of the LPSK need a tool that will help them create a binary configuration vector. Here we examine the design process for creating the configuration vector tool that creates the vector that specifies the initial secure state of the LPSK. The first section discusses the selection of the most appropriate programming language and associated tools necessary to design and implement the tool. The subsequent six sections discuss the thought-process involved in reaching an initial design. These sections begin with the breakdown of the configuration vector structure, followed by the emergence of the MVC model and basic design element ideas, and conclude with a final conceptual design sketch of the configuration vector tool.

A. OVERVIEW OF DEVELOPMENT TOOLS

With many different programming languages available, selecting the most appropriate language to implement the configuration vector tool was essential. Many of the most common programming languages provide GUI toolkits or Integrated Development Environment (IDE) GUI builders. These toolkits or IDEs help both professional and inexperienced developers quickly create professional-style applications. For the development of the configuration vector tool, the following languages were examined: Microsoft .NET [8], Apple Cocoa (Objective-C) [9], wxPython [10], and Java [11]. Since every language has its pros and cons, a selection process was devised in order to choose the most appropriate language for the configuration vector tool. The next section outlines the selection criteria and considerations.

1. Tool Selection Process

Six criteria were used to select the development language: MVC compliance, cross platform compatibility, online documentation, online tutorials and examples, available IDE GUI builders, and other developer considerations. First, each potential language was examined to see if it provided a framework for the MVC design paradigm. The next criterion examined each language for its ability to easily run on several standard operating systems with no modifications to either the operating system or the application.

The third and fourth criteria ensured that the language selected had sufficient online documentation, tutorials, and examples in order to guide and support the developer. The fifth criterion looked at the available IDEs with an integrated GUI builder in order to assist the developer in graphical design and implementation. The final criterion is the developer's experience and familiarity with the language. Table 1 summarizes the criteria used to select the language.

Criteria	Description
MVC Compliance	Does the language and associated IDE provide the library and tools necessary to easily create a MVC interface?
Cross Platform Compatibility	Can final application run on several standard operating systems with no modifications to either the application or the operating system?
Online Documentation	Is the online documentation sufficient and easily understood by an inexperienced developer?
Online Tutorials and Examples	Are online tutorials and examples readily available to the developer through both official and third party Web sites?
IDE GUI Builders	Do the IDEs available for the specific language include a comprehensive and refined GUI builder?
Developer Considerations	Is the developer familiar with this language?

Table 1. Candidate development language criteria

Table 2 shows a basic comparison of the candidate languages using the criteria described above. Based on the developer's limited experience and knowledge of Microsoft .NET and Apple Cocoa (Objective-C), these programming languages were immediately removed from the selection process. However, examination of both the Microsoft and Apple tools provided critical insights in design style and implementation methods that were used in future sections. The subsequent paragraphs detail the comparison of the two remaining languages: wxPython and Java Swing.

Selection Criteria											
Language	MVC	Cross Platform			Documentation / Support	GUI Builder / IDE				Developer	
		WIN	MAC	LIN		Name	\$\$	Description	Knowledge	Considerations	
Microsoft .NET	Yes	X			Excellent resources on the Microsoft Developers Network (MSDN). A vast amount of knowledge in the form of tutorials and examples.	Microsoft Visual Studio	\$0	An excellent tool used to develop console and GUI applications.	Very Limited	Other than Eclipse, a quality IDE does not exist for the Macintosh Platform	
						Eclipse IDE	\$0	A cross platform development tool. Does not have a built-in GUI builder, requires plugins, some of which are not free			
Apple Cocoa (Objective-C)	Yes		X		Excellent resources on the Apple developer connection website with tutorials and example code.	Apple Interface Builder	\$0*	Excellent interface builder that is intuitive and allows for the developer to easily create complex GUI layouts	Very Limited	*Both the Interface Builder and Xcode require the Macintosh Platform for development	
						Xcode	\$0*	An excellent IDE for creating Cocoa and other types of applications			
wxPython	Yes	X	X	X	Provides decent resources on the wxPython website. Unfortunately, some of the documentation is difficult to locate and even incomplete	wxGlade	\$0	A popular GUI toolkit. Not a full featured IDE but simply a designer. Used strictly for GUI builder	Moderate	The developer was the most knowledgeable in Python	
						Boa Constructor	\$0	A cross platform Python IDE and GUI Builder. Includes visual frame creation and manipulation			
						Python IDLE	\$0	No graphical interface. An excellent tool for creating python code.			
Java Swing	Yes	X	X	X	Java has excellent documentation and online support. There are excellent Java Swing Tutorials. There is also a plethora of online examples from various developers across the web.	Netbeans IDE	\$0	Includes a GUI builder that enables drag and drop of components from a palette to a canvas. Highly refined IDE	Moderate	The developer was familiar with the Java development process. Java has one of the largest documentation and example code bases around	
						Eclipse IDE	\$0	A cross platform development tool. Does not have a built in GUI builder, requires plugins, some of which are not free			

Table 2. Candidate development languages comparison

As discussed in Chapter II, Section B, the MVC design paradigm is an important consideration when selecting a design tool. Similar to the Java Swing discussion in Chapter II, wxPython combines the view and controller component into a single delegate. This allows a wxPython application to have the same flexibility as an application developed with Java Swing. Thus, both wxPython and Java comply with the MVC paradigm by implementing a modified MVC model.

Both wxPython and Java are also excellent candidates when cross-platform compatibility is an issue. Each is able to run natively on all of the major operating systems today (i.e., Windows, Macintosh, and Linux). wxPython is a wrapper for the wxWidgets cross-platform GUI library [12]. wxPython leverages the underlying operating system for graphical interface creation. This means that a wxPython application running on a Macintosh will use the Macintosh Aqua interface elements and appear to be a native Macintosh application. Likewise, the same wxPython application running on a Windows platform will use the Windows interface elements and appear to be a native Windows application. Java Swing, on the other hand, does not rely on a native operating system GUI component. This enables Java Swing to render its own components as necessary. Thus, the look and feel of Java Swing is completely platform independent; or stated differently, the GUI components rendered by Java Swing have the same appearance on all platforms.

The online documentation, tutorials, and available examples are a critical consideration for the selection of the design tool. The wxPython Web site [10] and wxPyWiki [13] have excellent resource and examples to help developers learn wxPython. Java Swing information is found at the Sun Microsystems Developer Network (SMDN) Web site [14]. On the SMDN site, there are countless tutorials and examples to guide developers in creating basic graphical applications. Although the wxPython documentation and support available online is sufficient, Java has much more information available from the official Java Web site as well as third party examples across the Internet. The information provided on the official Java Web site is also more refined than that found on the wxPython Web site. For an inexperienced developer, Java Swing provides more guidance.

The final major consideration is the available GUI builder development applications. wxPython provides a development application called PyCrust [10] as well as many examples of GUI components (PyCrust and examples are included with the standard wxPython installation) [10]. Unfortunately, PyCrust does not provide a graphical GUI creation tool. The example components, although extensive, do not help the developer in the placement of each component. Two other Python development applications are worth mentioning: wxGlade [15] and Boa Constructor [16]. wxGlade is a pure GUI designer that generates only GUI code [15]. Thus, wxGlade is not a full IDE. Developers looking for the traditional IDE support (i.e., inline compiling) must find a different tool. Boa Constructor fills this gap and is a complete Python IDE and wxPython GUI builder [16]. Boa Constructor provides a Python developer with visual creation of GUI components (drag and drop).

The Java Swing IDE is NetBeans. NetBeans is a refined IDE that features a Swing GUI Builder. The GUI builder allows the developer to create graphical components (e.g. buttons, tables, labels, etc.) by dragging and positioning the components on a canvas. This allows the developer to quickly create professional looking applications without spending the majority of time worrying about the “look and feel” of the interface. Although Boa Constructor is a decent GUI builder for Python, NetBeans for Java is more refined and intuitive.

Other considerations taken into account during the selection process were the developer’s knowledge and experience of each language as well as the ease of use of each of the development applications. The developer’s experience developing GUI applications was moderate and thus weighed heavily on the language selection. wxPython was initially chosen due to its ease of use in creating backend code for the interface and for the Boa Constructor IDE. However, as will be discussed in Chapter IV, wxPython was later dropped in favor of Java Swing and the NetBeans IDE.

2. Java Swing and NetBeans

Java Swing was selected as the development tool for the configuration vector tool. Specifically, the Sun Microsystems Java Development Kit (JDK) SE 6 Update 16 for

Mac OS 10.6 was selected as the development platform. Java was chosen for its extensive libraries, excellent online support and tutorials, cross-platform compatibility, and its Swing GUI capabilities. As a result of its portability, any system supporting the Java Runtime Environment should be able to compile and run the configuration vector tool.

While several of the Java files for the model component of this thesis were written using a standard text editor, the view and controller components (i.e., the GUI) took advantage of the NetBeans IDE version 6.7 [17]. The NetBeans IDE is a free open-source IDE that runs on Windows, Linux, Mac OS X, and Solaris. The NetBeans IDE features a Swing GUI Builder (formerly Project Matisse) that allows developers to create professional looking applications. Leveraging Java as the development environment, the next section outlines the configuration vector structure. Understanding this structure is critical to implementing the configuration vector tool.

B. CONFIGURATION VECTOR FORMAT

As discussed in the background section, the LPSK configuration vector is the crucial component for initializing the LPSK. In order to create a GUI application that creates a binary vector, it is important to examine the format of the vector. By examining the format of the vector, it is possible to find natural divisions based upon the structure and semantics of the configuration vector that hint at possible GUI designs. The remainder of this section will examine the vector structure and attempt to show the natural divisions in the structure that were used to create a GUI.

1. Basics

The binary configuration vector is defined in a C header file—`lpsk.h` (see Appendix). The header file outlines the complete structure of the configuration vector. The first portion of the header file is the definitions section. This section defines the constants used throughout the file. After the definitions section, the header file contains many different C structs. *Struct* is short for *structure*, or user-defined data type. A struct in the C programming language is like a class in Java but without methods. It is used to

logically bundle or package related data. A struct aggregates many different data types into a single user-defined data type. The C programming language does not limit the size or type of any of the objects contained in the struct. This means that a struct can package other structs.

Another important element of the header file is *arrays*. Arrays in the C programming language are variables that store multiple items of the same data type. A one-dimensional array is ordered via the index of each item in the array. Individual items contained in an array can be read or written by referencing the specific index of the array. For example:

```
array_demo[] = {a, b, c};  
array_demo[1] = b;
```

A two-dimensional array is an array inside of an array—a matrix. Thus, a two-dimensional array contains rows and columns of data. Similar to one-dimensional arrays, values within two-dimensional arrays can be read or written by referencing the specific index of the array. For example:

```
array_demo[][] = { {a, b, c}, {d, e, f} };  
array_demo[1][1] = e;
```

For the purpose of this work, structs may contain any declared or standard data type, such as integers and floating point variables. With this basic knowledge of the LPSK header file, the next section gives a detailed breakdown of the configuration vector structure into its basic components.

2. Structure Overview

The configuration vector is specifically located in the `vector_struct` struct of the header file. The `vector_struct` struct (see Figure 6) contains the format version, structure magic number, the number of partitions, the Trusted Path Application (TPA) partition identifier, the number of eventcounts, the number of sequencers, a partition-to-partition permissions two-dimensional array, and six additional structures. The six structures are the partition definition struct, data segment struct, memory segment

struct, eventcount struct, sequencer struct, and the subject-resource permissions struct.

The exact names of the structures are listed below:

- `partition_struct partitions[]`
- `datafile_struct datafile[]`
- `memory_struct memory[]`
- `synchronization_struct eventcounts[]`
- `synchronization_struct sequencers[]`
- `subj_res_perm_struct subj_perm[]`

```
// The following structure is an LPSK configuration vector
//
typedef struct {
    unsigned int    version;                // The format version
    unsigned int    magic;                  // The structure magic #
    unsigned int    num_partitions;         // The # of partitions
    int             tpa_partition;          // The TPA partition
    partition_struct partitions[MAX_PARTITIONS]; // All the partition defn's
    datafile_struct datafile[MAX_DSEGS];    // additional data segments
    memory_struct   memory[MAX_MSEGS];     // additional memory
    unsigned int    num_eventcounts;        // The # of eventcounts
    synchronization_struct eventcounts[MAX_EVENTCOUNTS]; // eventcount data
    unsigned int    num_sequencers;        // The # of sequencers
    synchronization_struct sequencers[MAX_SEQUENCERS]; // sequencer data
    // Partition -> Partition permissions
    unsigned int    part_perm[MAX_PARTITIONS][MAX_PARTITIONS];
    // subject -> resource permissions
    subj_res_perm_struct subj_perm[MAX_SUBJECTS];
} vector_struct;
```

Figure 6. Configuration vector excerpt from `lpsk.h`

Several of these six structures contain additional structures embedded within them. Thus, the larger structure is composed of smaller building blocks, which enhances its analyzability and decreases its ultimate complexity. Figure 7 captures the top-level structure (i.e., the configuration vector) from the `lpsk.h` file.

As seen in Figure 7, `vector_struct` is the base struct containing all the other structs. Each sub-structure is a one-dimensional array of structs. This means that one sub-structure may have multiple structs contained within it. Each sub-structure array uses the constant values defined in the definitions section of the header file to set the

maximum number of structs for the given sub-struct array. For simplicity, the remainder of this section will drop the constants from the discussion and only refer to the array of structs as a struct.

From Figure 7, `partition_struct` holds `process_struct` and the process struct holds the `subject_struct`. This figure also shows that the `eventcounts struct` and `sequencers struct` are simply represented by a synchronization struct. By looking at this code, one can see that the natural breaks in the overall vector structure are the structs. Following this logic, a Java class can represent each struct.

Another important piece of the vector structure are the `perms[]` one-dimensional arrays (see Appendix) and the `part_perm[][]` two-dimensional array. These are important because they required special treatment when designing the GUI. Fortunately, the `perms[]` arrays are very similar throughout the `lspk.h` file—only differing with regard to the length of each array. This means that a single Java class may be used to represent the arrays. For the two-dimensional array, a separate Java class was sufficient to handle it.

The final pieces of the puzzle are the definitions of the constants that define the number of elements of the various structures (i.e., from Figure 6, `MAX_PARTITIONS`, `MAX_DSEGS`, etc.). These are defined in the definitions portion of the `lpsk.h` file (see Appendix) and are omitted in Figure 7 for simplicity. A single Java class easily captures these definitions.

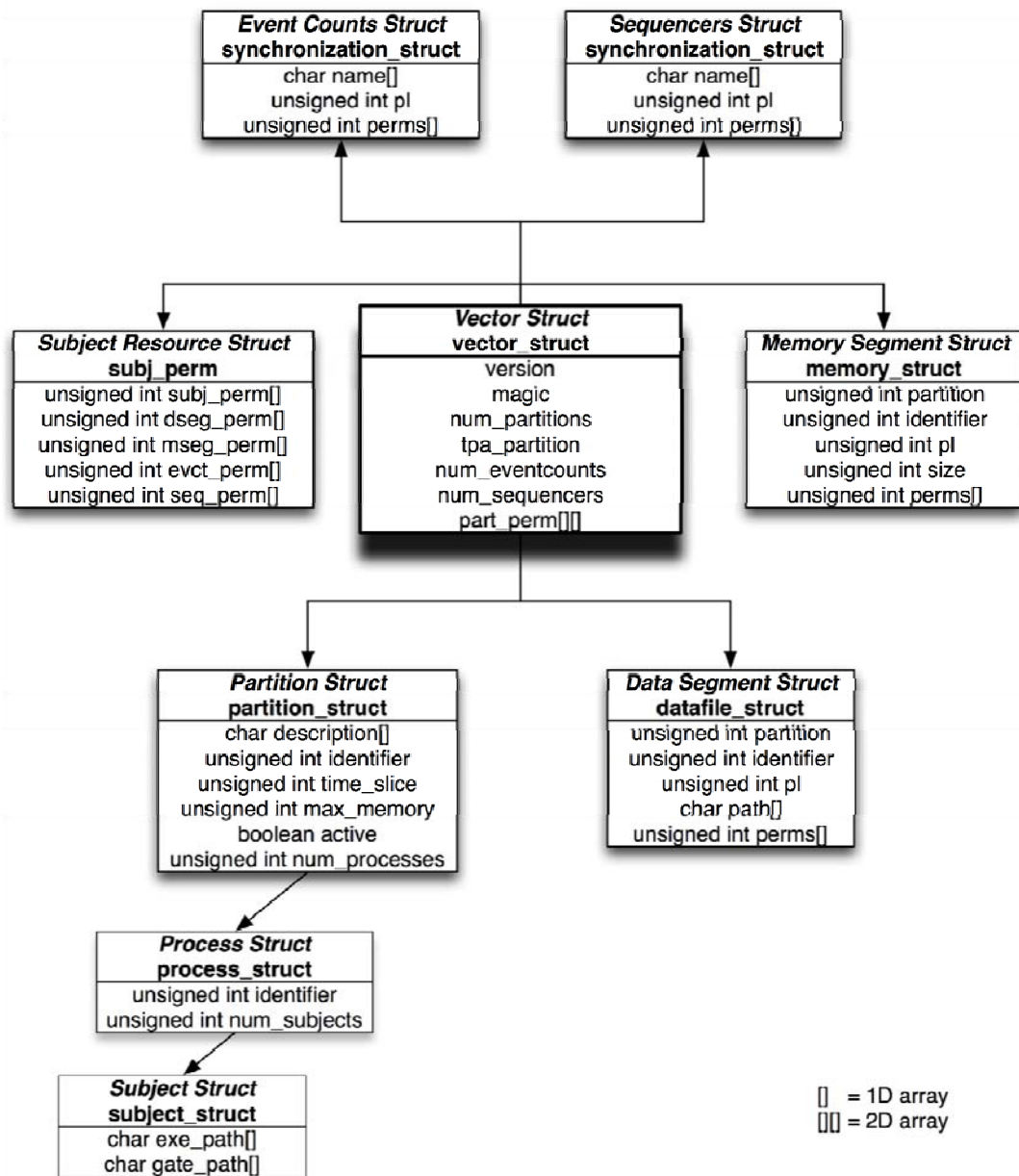


Figure 7. Configuration vector structure diagram based on the lpsk.h

This section showed the vector structure defined in the lpsk.h file. The next section expands upon this structure as it relates to the MVC paradigm.

3. MVC Model Component

By understanding the structure of the specific structs in the `lpsk.h` file, the model component of the MVC paradigm emerges. Using Figure 7 as a guide, the structures of the `lpsk.h` file can be translated into individual Java classes. These Java classes represent the model component of the MVC. Each of these Java classes is discussed in detail later during the discussion of the configuration vector tool implementation.

At this point, it is worth noting the importance of the model component of a GUI application. A robust model capable of containing data gathered by a GUI is critical to application success. In addition, if the model component is created correctly, then any GUI could be used to capture the data from the user. This allows any future changes to the user interface, including a complete overhaul of it, to be accomplished relatively easily without affecting the underlying model.

This section outlined basic knowledge for understanding the `lpsk.h` file and then showed the logical breakdown of the individual components of the configuration vector. Using the logical breakdown also led to the emergence of the model component described by the MVC paradigm. The next section also will use Figure 7 to discuss the primary graphical design element choice, which provides the foundation for the graphical aspect of the configuration vector tool: the view/controller component.

C. THE PRIMARY GRAPHICAL DESIGN ELEMENT

Not only did Figure 7 help with the design of the underlying model component of the GUI, but it also provided a conceptual breakdown of the data for the view component. Since the structs naturally divide the `vector_struct`, a possible graphical representation of the data is a table. In this case, a table can graphically represent the data from each structure. Since the `vector_struct` incorporates six structs, the main application window was designed to point to at least six different tables. Each of these tables contains the attributes necessary to create each struct. For instance, an eventcounts table must contain at least the name, privilege level, and permissions attributes corresponding to the attributes of the synchronization struct. Figure 8 shows the

transformation of the `vector_struct` attributes to the table column headers. The next section shows the modifications to this table and the incorporation of all elements from the `vector_struct` defined in the `lpsk.h` file into the graphical representation.

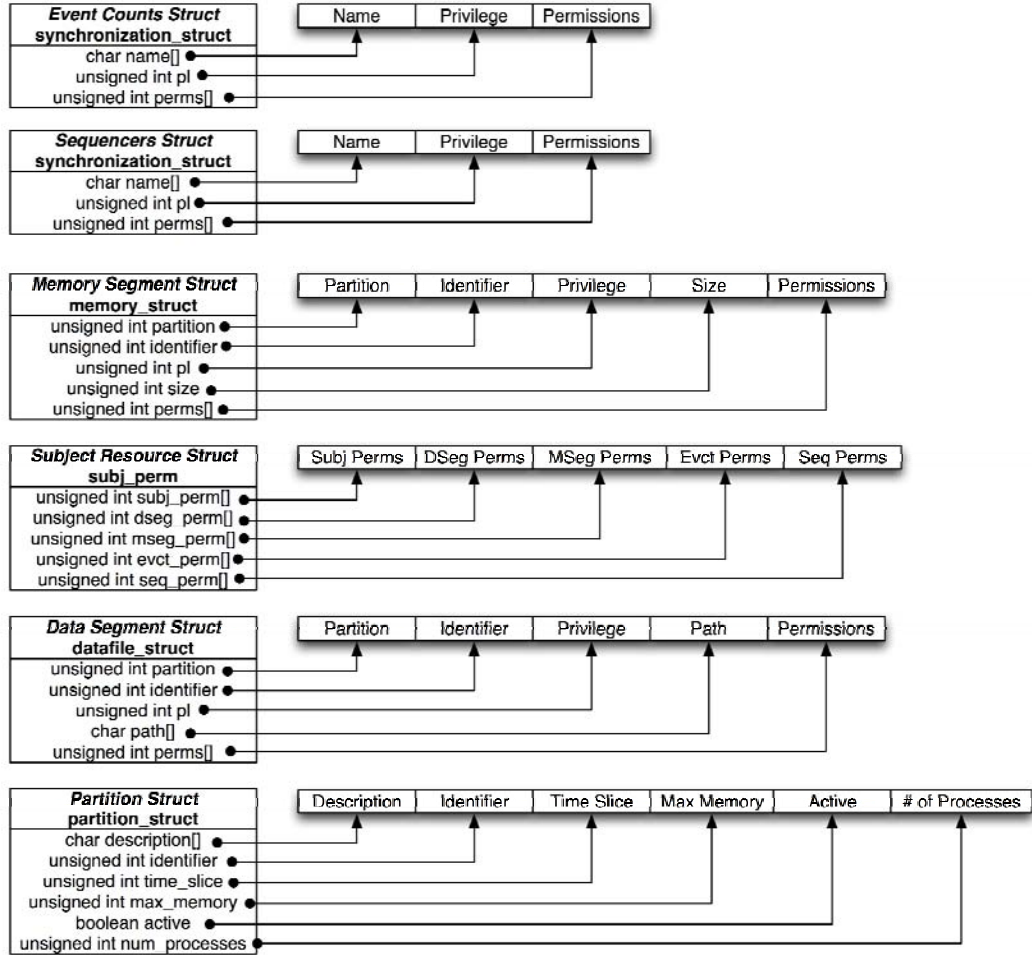


Figure 8. Illustrates the transformation of the `vector_struct` attributes to table column headers

D. DESIGN REFINEMENT

This section refines the conceptual graphic design to incorporate all attributes from the `vector_struct` and its sub-structures. The majority of this section will use graphics to illustrate the thought process behind the refinements. In the previous section, Figure 8 showed the initial transformation of the `vector_struct` sub-structs into the

individual table column headers. The next step incorporates the main vector attributes into this graphical concept.

The `vector_struct` attributes contain the tables described in Figure 8 and seven additional attributes (i.e., `version`, `magic`, `tpa_partition`, `num_partitions`, `num_eventcounts`, `num_sequencers`, and `part_perm[][]`). Since the additional attributes are applicable for the entire vector struct, all of these attributes were added to a window panel. The entire `vector_struct` became the main application window. The tables from Figure 8 then became a tabbed panel within a separate window panel. Figure 9 depicts this evolution of the conceptual design. In Figure 9, the table column headers are placed inside a main application window. Thus, the main application window represents the top-level structure (i.e., the `vector_struct`). The table column headers represent the sub-structures of the `vector_struct` and were collapsed into a tabbed panel with one table per panel. The individual attributes of the `vector_struct` were incorporated into the vector attributes panel of the main application window.

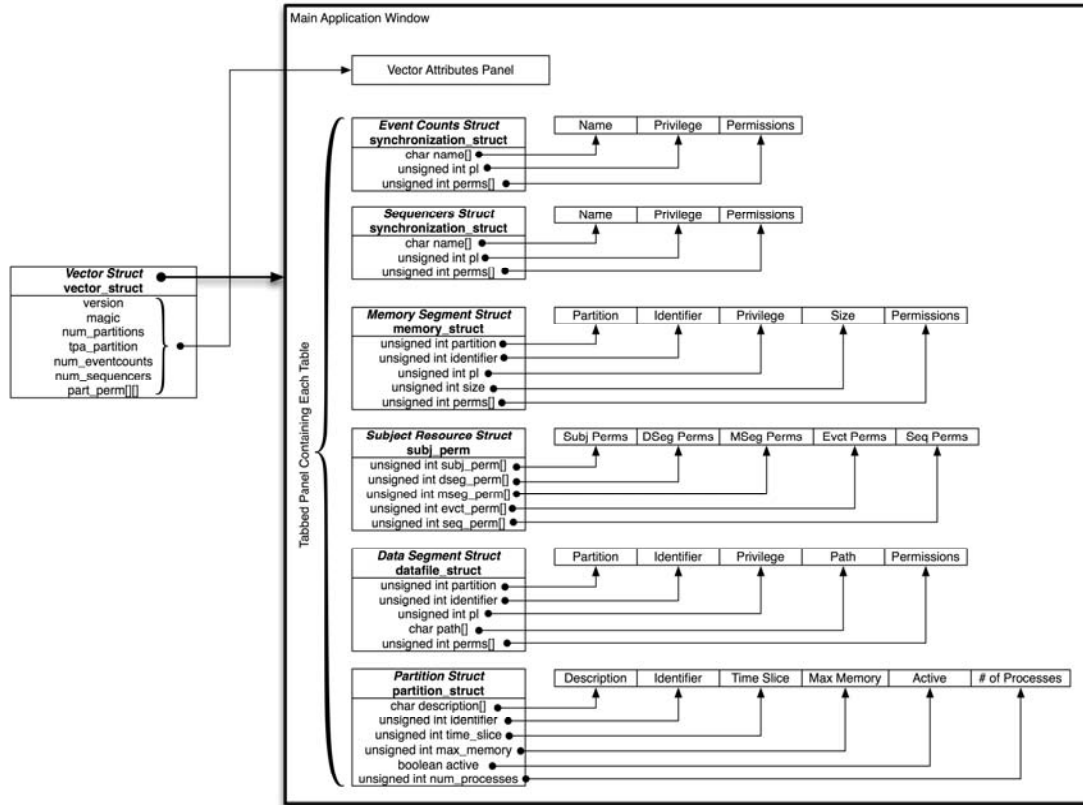


Figure 9. Illustration depicting the incorporation of the tables and `vector_struct` attributes into a main application window

The next step involved adding the sub-structs of the `partition_struct` to this design (i.e., the `process_struct` and the `subject_struct`). Since both the `process_struct` and `subject_struct` have features similar to those of other structs previously described, a logical approach was to treat each of these as a table within its parent table. In other words, the partition table contains a process table and the process table contains a subject table. Figure 10 shows the addition of an extra column to the partition table to account for the linkage to the process table. Similarly, the process table contains an additional table column not listed in its struct for the subject table. Figure 10 illustrates the nesting of one table inside a parent table. Specifically, the process table was linked in its parent (i.e., partition table) and the subject table was linked in its parent (i.e., process table).

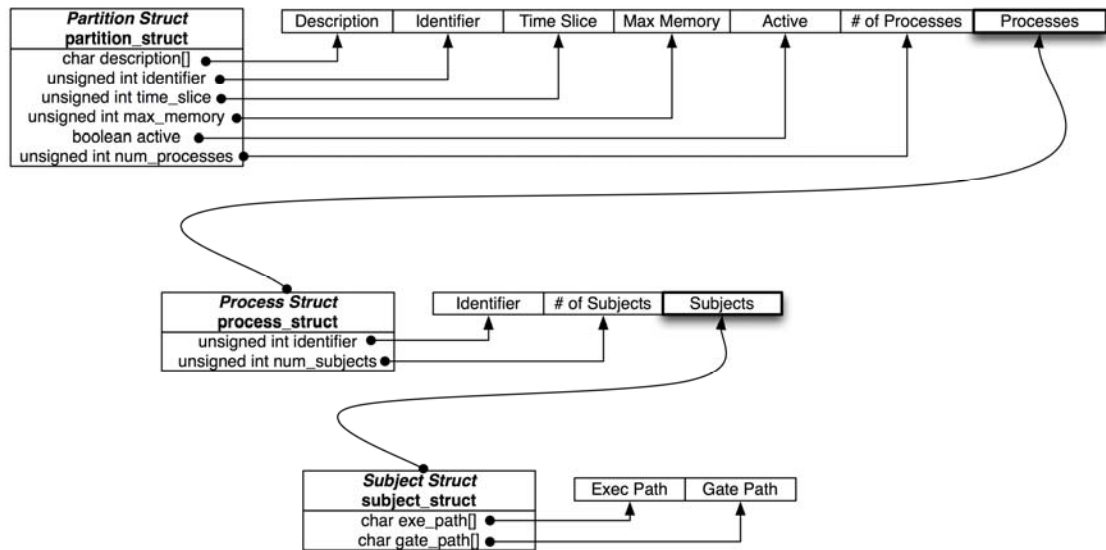


Figure 10. Illustration of the nesting child tables within parent tables

Unfortunately, although the design seems to capture the complete `vector_struct`, there are several instances where more refinement was necessary. First, placing the `part_perm[][]` two-dimensional array attribute of the `vector_struct` in the vector attributes panel with other attributes of completely different data types seemed awkward and inconsistent. A more elegant solution was necessary. In order to refine the `part_perm[][]` attribute, it was necessary to remove it from the vector attributes panel. Since the `part_perm[][]` attribute contains partition-to-partition permissions for each partition, it made sense to add it as an additional table within the tabbed table panel of the main application window. Figure 11 depicts this design decision by showing similar data type objects in the vector attributes panel and moving the `part_perm[][]` two-dimensional array to be within the tabbed table panel of the main application window.

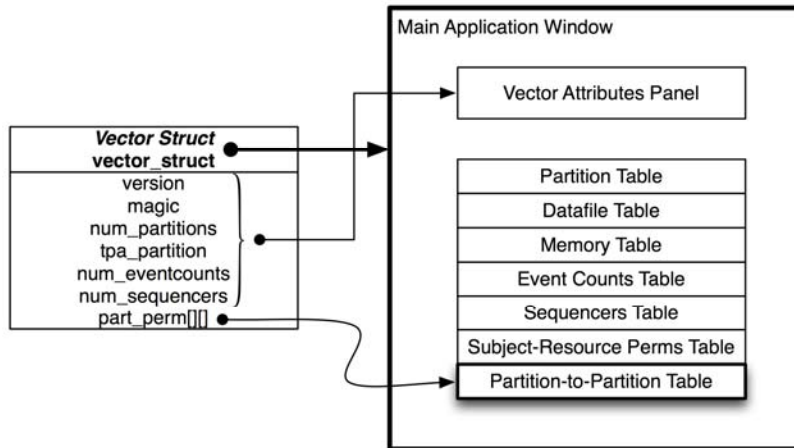


Figure 11. A refinement of the initial window design showing the addition of the partition-to-partition table

Second, the subject and process embedded tables needed to be refined to better capture the required data. Thus, the attributes of the `process_struct` and `subject_struct`, were combined into a single entity. Since the `process_struct` contains little information besides the associated subject table, the table was transformed into a process window. This new process window captures the necessary data similar to the way the main application captures its data—a window panel to capture the process attributes (i.e., identifier and number of subjects) in a processes attribute panel and provides a table to capture the specific subject information. Figure 12 describes this transformation.

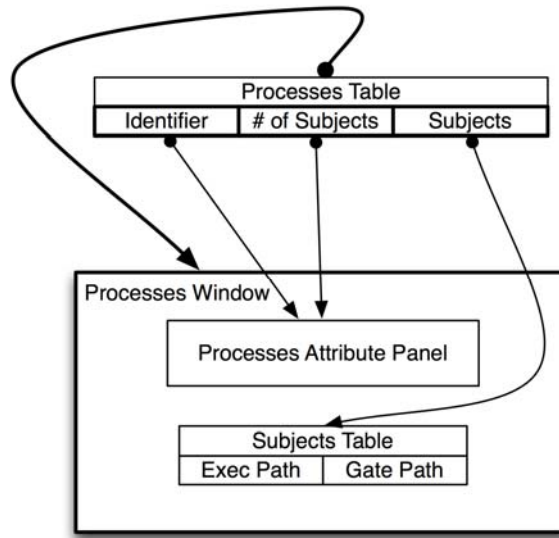


Figure 12. A refinement of the process and subject tables. The two tables are merged into one window

Finally, the permissions column of all of the main tables described in Figure 8 is repeated in several tables. This repetition among the tables requires special consideration. In order to correctly capture the required data, a common interface across all tables with the permissions column must be created. Thus, this refinement to the design involved creating a common permissions window interface for each of the `perms[]` attributes of the `vector_struct` sub-structures. In this case, each of the columns of each table is linked to a new window that contains a sub-table—a permissions table. Figure 13 shows the relationship from the table column headers to the new permissions table. It is important to note that each permissions column associated with the various tables receives a new window upon request. Thus, a permissions window originating from the `evencounts` table is not the same as a window originating from the `sequencers` table (this is true for all instances of permissions tables). Only the visual representation of this new window is common across all tables (i.e., a common interface with different values for all tables).

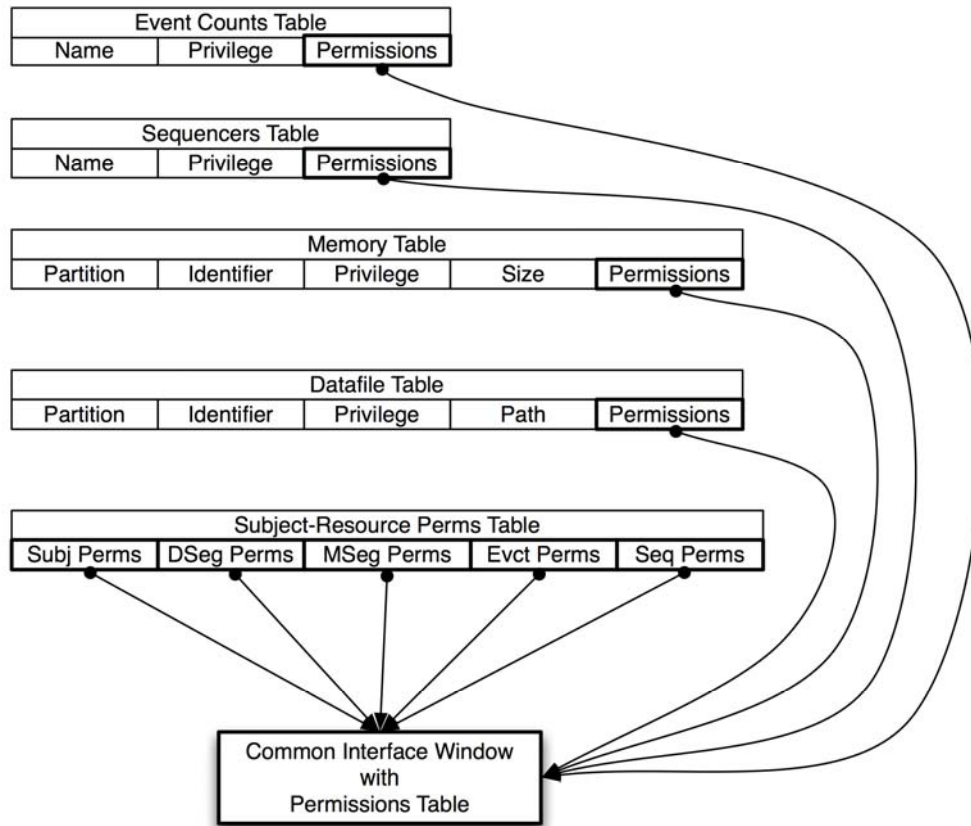


Figure 13. All permissions columns within each table is represented by a common interface window

Figure 14 shows the complete view of the tables and their associated attributes. One minor refinement completed the design. Since each row of a table represents an instance of the table's struct (i.e., a single row of a table represents a single element contained in its struct array), it was useful to add the appropriate label for each table that signifies what each row for each table represents. For instance, a partition number column was added to the partition table. Similarly, a datafile number column was added to the datafile table. This is consistent across all tables except the partition-to-partition permissions table, subjects table, and permissions table. The partition-to-partition permissions table shows the inter-partition permissions. Figure 14 illustrates the addition of the identification label columns (discussed above) to each table. The figure also shows the hierarchical relationship of the tables. Since all of the tables shown in Figure 14 are subordinates of the `vector_struct`, the tables with solid dots are considered

intermediate tables and the two tables at the end of arrows are considered leaf tables. The object number of the permissions leaf table will change based on the intermediate table from which it originated. In other words, a permissions leaf table originating from the datafile table will show the datafile number in place of the object number. Finally, the subjects table embedded in the process table has a privilege level added to distinguish between individual subjects within a particular process.

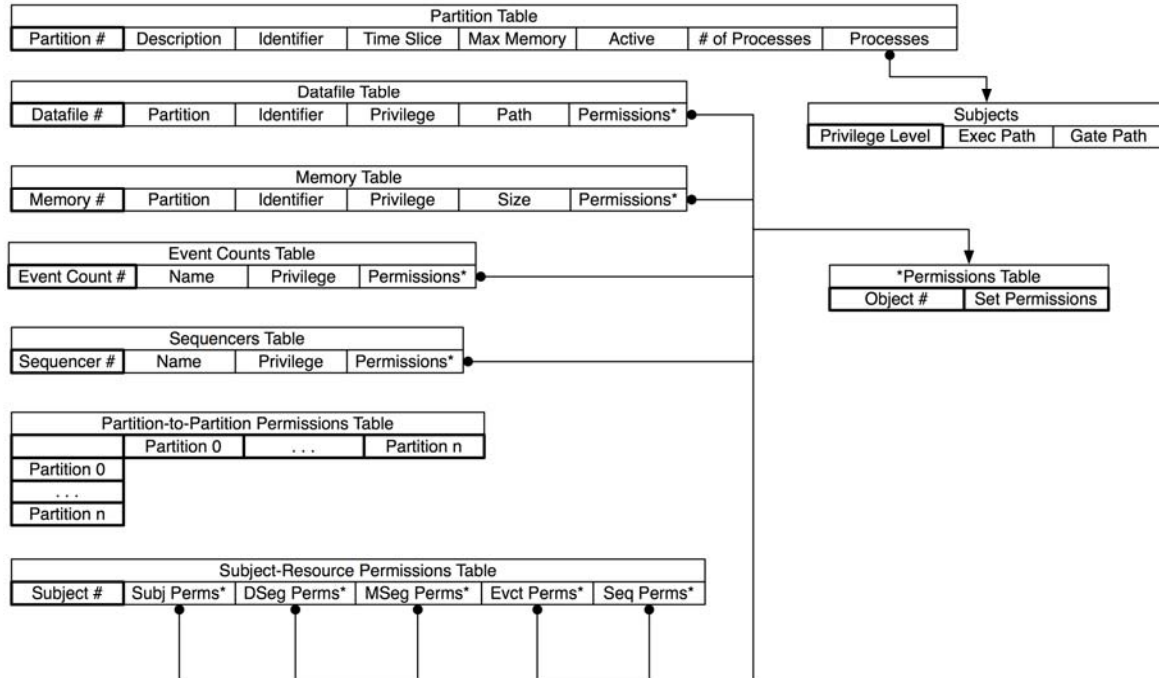


Figure 14. Final table column headers as defined by the `lpsk.h` file

The final figure in this section, Figure 15, shows the complete design concept. The main application window contains the vector attributes panel and a tabbed tables panel. The vector attributes panel contains all of the attributes of the `vector_struct` except the sub-structures. The tabbed tables panel contains the sub-structures of the `vector_struct`. There are two other windows depicted in this figure that appear upon selecting the appropriate column in the respective main tables. A process window appears when the process column is selected in the partition table. The appropriate permissions window of the originating table appears when the permissions column is selected.

The figures in this section showed the refinement of the primary design element (i.e., the table) as well as the overall design concept for the configuration vector tool GUI. Once defined, it is necessary to determine the requirements for the configuration vector tool. The next section discusses the specific requirements for the configuration vector tool application.

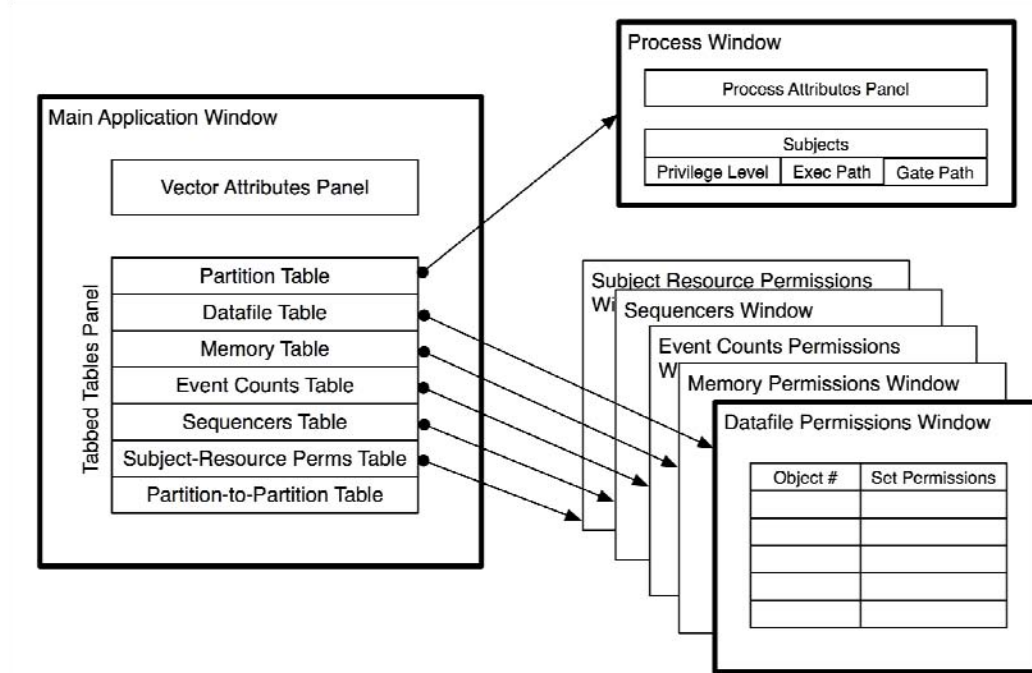


Figure 15. A complete design concept of the `vector_struct`

E. CONFIGURATION VECTOR TOOL REQUIREMENTS

As discussed in Chapter II, a configuration vector exists in two formats: a human-readable format or a binary format. The configuration vector tool is an application that creates valid configuration vectors (both human-readable and binary) and converts a configuration vector from one of these formats to the other. This is the overarching requirement for this research. The remainder of this section outlines the basic requirements for the configuration vector tool.

1. Basic Requirements

The configuration vector tool must be an offline software application that is able to execute on a standard operating system. The tool must have the ability to take as an

input a human-readable configuration vector and produce the equivalent binary configuration vector. The tool must also have the ability to take as an input a binary configuration vector and produce the equivalent human-readable configuration vector. The user shall have the ability to select the output file name and destination for both the generated file formats. All generated files, whether binary or human-readable, must be syntactically correct. The list below shows these basic requirements.

- Remove underlying complexities associated with configuration vector creation
- Offline application (i.e., a stand-alone application that requires no additional LPSK software)
- Executes on a standard operating system
- Reads syntactically correct binary configuration vector file
- Reads syntactically correct human-readable configuration vector file
- Writes a syntactically and semantically correct binary configuration vector file
- Writes a syntactically and semantically correct human-readable configuration vector file or only a syntactically correct human-readable configuration vector file

Before continuing, it is useful to define syntactically correct. A *syntactically correct* file conforms to the rules and structure defined for the configuration vector file. A syntactically correct configuration vector may fail a semantic (variable bounds case only) check. If a file passes the syntax check but fails a bounds check, then the latter check is considered a semantic failure. Thus, failing a semantic check means a variable was outside the defined upper or lower limit. References to a semantic check refer only to the bounds or limits of the variables defined in the configuration vector file.

Based on the requirements stated above and the definitions of both syntactic and semantic correctness, Figure 16 shows the configuration vector tool state transition diagram. The input and output configuration vectors shown in the figure can either be in binary or human-readable form. Table 3 represents the possible output configuration vector file types given a specific input configuration vector file type. A human-readable

file read into the tool can write out a binary or human-readable file. A binary file read into the tool can write out a binary or human-readable file. The *none* input in the table refers to the creation of a new file from the tool.

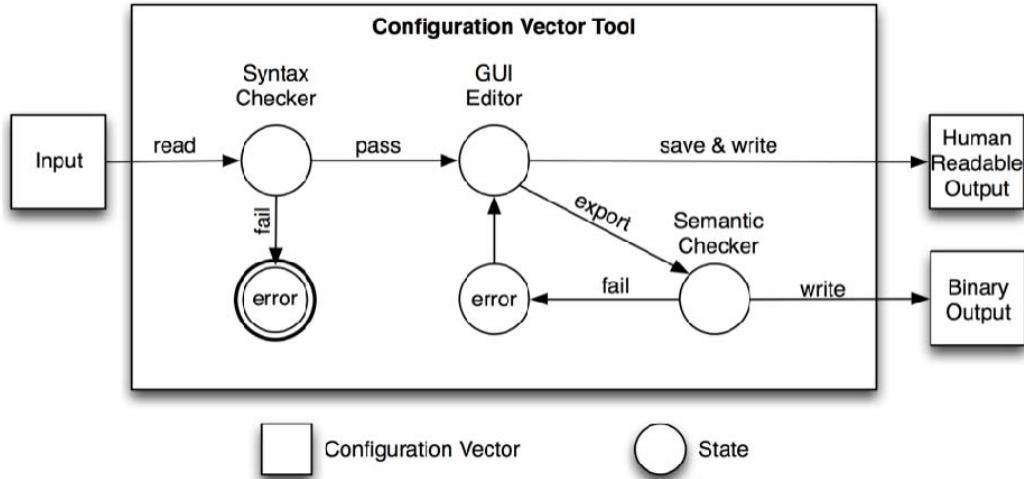


Figure 16. Configuration vector tool state transition diagram

Configuration Vectors	
Read In (Input)	Write Out (Output)
Human-readable	Binary
Human-readable	Human-readable
Binary	Binary
Binary	Human-readable
None (new file)	Binary
None (new file)	Human-readable

Table 3. The possible output file type given a specific input file type

A configuration vector input file read into the tool must first enter the syntax checker state. The syntax checker ensures that the input file conforms to the configuration vector file standard. If the input file fails this check, the syntax checker reports an error to the user. If the input file passes, it is sent to the GUI editor state. The GUI editor state allows the user to make changes to the vector file. Upon completion of changes (if any), the user has two options: save or export the vector file. If the user chooses to save the vector file, it is written to a human-readable file without applying a semantic check (giving users the flexibility of saving unfinished vectors that are probably not ready for any verification). If the user chooses to export the vector file, the tool

progresses to the semantic checker which checks the bounds of the variables contained in the vector. If the file fails in this state, the semantic checker reports an error to the user and sends the file back to the GUI editor state. If the file passes the semantic checker, then a syntactically correct and semantically correct configuration vector file may be written to disk. In both cases, saving or exporting, the editor prevents writing a vector file with incorrect syntax to disk. Thus, it is not necessary to check for syntactic correctness during the output phase.

The next section outlines the specific details of a syntactically correct configuration vector file. The section also outlines the semantic tests for each of the objects contained within a configuration vector file.

2. Detailed Requirements

The detailed requirements for the configuration vector tool are described in this section. This section mostly contains lists of tables and values that outline the upper and lower bounds of all variables currently defined in the `lpsk.h` file. The first table of this section, Table 4, shows the description, name, and value of the constants defined in the `lpsk.h` definitions section. These values may change in the future and are listed only as a reference for the remainder of the section. Thus, the configuration vector references the constant name rather than the value.

Maximum Constant Values		
Description	Name	Value
Maximum length of description string	MAX_DESC	32
Maximum length of exported object name	MAX_NAME	32
Number of privilege levels supported by the CPU	NUM_PLS	4
Maximum number of datafile segments	MAX_DSEGS	64
Maximum number of memory segments	MAX_MSEGS	32
Maximum path length for file names	MAX_PATH	64
Maximum number of partitions	MAX_PARTITIONS	8
Maximum number of processes (per partition)	MAX_PROCESSES	1
Maximum number of subjects (per process)	MAX_SUBJECTS	24 [†]
Maximum number of event counts (synchronization struct constant)	MAX_EVENTCOUNTS	32
Maximum number of sequencers (synchronization struct constant)	MAX_SEQUENCERS	32

$$^{\dagger} (\text{MAX_PARTITIONS} * \text{MAX_PROCESSES} * (\text{NUM_PLS} - 1))$$

Table 4. Constant values defined by the `lpsk.h` file

The second set of requirements, shown in Table 5, is simply a mapping table that shows how the permissions are mapped from the human-readable format to the binary format. The permission mapping for all permissions scattered throughout the structs reference this table.

Permissions		
Description	Human-readable format	Binary format
No Access	NA	0
Read only	RO	1
Read/await eventcount		
Read and write	RW	2
Signal subject		
Read/await/advance eventcount		
Ticket sequencer		
Signal subject	WO	3
Advance eventcount		

Table 5. Mapping of permissions from the human-readable format to the binary format

The third set of requirements is shown by the structs bound requirements table, Table 6. This table is a set of tables that outlines the upper and lower bounds for each struct of the `lpsk.h`. The table is broken into the eight structs that make up the `vector_struct`.

Structs Bounds Requirements			
vector_struct			
Description	Name	Type	Bounds
The format version	version	unsigned int	≥ 0
The structure magic #	magic	unsigned int	N/A (read-only)
The # of partitions	num_partitions	unsigned int	(0, MAX_PARTITIONS)
The TPA partition	tpa_partition	int	[0, MAX_PARTITIONS)
The # of eventcounts	num_eventcounts	unsigned int	[0, MAX_EVENTCOUNTS)
The # of sequencers	num_sequencers	unsigned int	[0, MAX_SEQUENCERS)
Partitions structs	partitions	partition_struct[]	[0, MAX_PARTITIONS)
Datafile structs	datafile	datafile_struct[]	[0, MAX_DSEGS)
Memory structs	memory	memory_struct[]	[0, MAX_MSEGS)
Eventcounts structs	eventcounts	synchronization_struct[]	[0, MAX_EVENTCOUNTS)
Sequencers structs	sequencers	synchronization_struct[]	[0, MAX_SEQUENCERS)
Partition to partition permissions two-dimensional array	part_perm	unsigned int[][]	[0, MAX_PARTITIONS) [0, MAX_PARTITIONS)
Subject resource permissions	subj_perm	subj_res_perm_struct[]	[0, MAX_SUBJECTS)
partition_struct			
Description	Name	Type	Bounds

Structs Bounds Requirements			
Description of the partition	description	char[]	[0, MAX_DESC)
Partition identifier	identifier	unsigned int	≥ 0
Fixed scheduling time sliced	time_slice	unsigned int	[0, 100]
Maximum memory a partition can use	max_memory	unsigned int	$> 0^{\dagger}$
Active or passive partition	active	boolean	Active = TRUE Passive = FALSE
Number of processes in the partition	num_processes	unsigned int	[0, MAX_PROCESSES]
Process structs	processes	process_struct[]	[0, MAX_PROCESSES]

datafile_struct			
Description	Name	Type	Bounds
Partition to load in	partition	unsigned int	[0, MAX_PARTITIONS]
Datafile identifier	identifier	unsigned int	≥ 0
Privilege levels to load in	pl	unsigned int	[0, NUM_PLS)
Location of the datafile on the disk	path	char[]	[0, MAX_PATH)
Permissions for each partition	perms	unsigned int[]	[0, MAX_PARTITIONS)

memory_struct			
Description	Name	Type	Bounds
Partition to load in	partition	unsigned int	[0, MAX_PARTITIONS)
Datafile identifier	identifier	unsigned int	≥ 0
Privilege level to allocate memory segment in	pl	unsigned int	[0, NUM_PLS)
Size of the requested memory segment	size	unsigned int	$> 0^{\dagger}$
Permissions for each partition	perms	unsigned int[]	[0, MAX_PARTITIONS)

synchronization_struct (eventcounts and sequencers)			
Description	Name	Type	Bounds
Name of object	name	char[]	[0, MAX_NAME)
Privilege level of object	pl	unsigned int	[0, NUM_PLS)
Permissions for each partition	perms	unsigned int[]	[0, MAX_PARTITIONS)

subj_res_perm_struct			
Description	Name	Type	Bounds
Other subjects	subj_perm	unsigned int[]	[0, MAX_SUBJECTS)
Data segments	dseg_perm	unsigned int[]	[0, MAX_DSEGS)
Memory segments	mseg_perm	unsigned int[]	[0, MAX_MSEGS)
Eventcounts	evct_perm	unsigned int[]	[0, MAX_EVENTCOUNTS)
Sequencers	seq_perm	unsigned int[]	[0, MAX_SEQUENCERS)

Structs Bounds Requirements

process_struct			
<i>Description</i>	<i>Name</i>	<i>Type</i>	<i>Bounds</i>
Process identifier	identifier	unsigned int	≥ 0
Number of subjects in the process	num_subjects	unsigned int	[0, NUM_PLS]
Subject structs definition	code	subject_struct[]	[0, NUM_PLS)

subject_struct			
<i>Description</i>	<i>Name</i>	<i>Type</i>	<i>Bounds</i>
Location of executable file	exe_path	char[]	[0, MAX_PATH)
Location of gate information	gate_path	char[]	[0, MAX_PATH)

[†] Limited by physical memory of the system

Table 6. Upper and lower bounds for all objects contained within every struct of the vector_struct

The final set of requirements for the configuration vector tool is captured in Table 7. The requirements in this table show the dependencies of some of the objects contained in the configuration vector. The table also clarifies some requirements from the previous table. The next section outlines the informative message reported to the user upon reaching an error state.

Additional Requirements		
Applies to	Name	Requirement
All Structs	identifier	All identifiers are greater than zero and unique relative to a respective struct.
	PI	Always have the values of 0, 1, 2, or 3
	perms	Binary values always 0, 1, 2, or 3; corresponding to the human-readable values of NA, RO, RW, WO respectively
Partition Struct	active	At least one partition in the set of partitions must be active
		All active partitions must have must have 1 process
		A passive (inactive) partition cannot have any processes (and thus no subjects)
		A passive (inactive) partition must have a time slice of 0
	time_slice	Individual fields must be greater than or equal to 0 and less than or equal to 100
		The total of all time slices across the partition set must sum to 100
	max_memory	Must be greater than 0
Datafile Struct	partition	Must reference a defined partition
Memory Struct	partition	Must reference a defined partition
Sequencer Struct	perms	Can only be NA or RW (this supersedes the perms requirement stated above)
Subject Struct	exe_path	PL0 must contain the path to the LPSK kernel. Unfortunately, there is currently no way to verify this requirement. Thus, the exe_path is only checked to ensure that the path is within the size constraints for a path.
	Gate_path	PL3 must be empty. The gate_path is also checked to ensure that the path is within the size constraints for a path.
Process Struct	num_subjects	Valid subject_struct exe_path is a subject. Thus, the minimum number of subjects is always 1 (the PL0 subject) and the maximum is 4.
Vector Attributes	tpa_partition	The tpa_partition must be an active partition as well as a defined partition.

Table 7. Dependency relationships of configuration vector fields

3. Error Message Requirements

Since the configuration vector tool must ensure that it generates a syntactically correct and semantically correct output file, informative error messages are necessary when the user provides invalid values. The first error message that shall be reported to the user occurs when reading an invalid (syntactically incorrect) input file (see Figure 16). Since this error is syntactic in nature, the tool should simply return an error stating the input file is syntactically incorrect and the line number of the first error encountered

by the tool. The tool will report this error to the user as a popup window dialog. After the user accepts this error message, the tool will load a blank vector and display the main window.

The next error that could occur is a semantic (out of bounds) error. Two specific instances of this type of error require explanation. The first type of error occurs when the user attempts to open a syntactically correct, but semantically, incorrect input file. In this case, the tool will allow the input file to be loaded into the GUI editor with a warning dialog pointing out the error(s). This will allow the user to edit this input file. After the user finishes an editing session, the user has two choices: save or export the vector file. If the user chooses to save the vector, a human-readable file is written to disk with only basic error checking (i.e., the bounds of the fields are checked but the dependency of one field to another are not checked). Saving a vector file allows the user to skip complicated error checking and keep vector files that still require work. However, if the user chooses to export to a binary format, the tool must check the values before writing out a syntactically and semantically correct file. If the file still has semantic errors, the tool will report a semantic error message and the specific item in the vector that failed the check and return to the editor.

The second type of error is similar to the first except the user does not attempt to read-in an input file. If a user creates a new vector, the new vector cannot be written out to disk as a binary output file until it is error free. All steps for this case are the same as the first case.

This section outlined the basic requirements as well as detailed requirements for each individual struct. It also provided a generic state transition diagram showing the progression of an input configuration vector file through the configuration vector tool to the final output configuration vector file. Finally, the section outlined the error messages that should be reported to the user if the configuration vector is syntactically or semantically incorrect. The next section discusses the feature set of the configuration vector GUI that meets these requirements.

F. CONFIGURATION VECTOR TOOL FEATURE SET

Feature cascade is a term used in the design of software applications where a simple application can quickly become complex due to the addition of many features that are not relevant to the original intent of the program [18]. The addition of features not only increases the overall complexity of an application, it also tends to negatively affect the application performance and make it more difficult to use. Applications with many features are not necessarily the best applications. Applications with the necessary, sufficient, and appropriate features to meet the original intent and requirements usually are the most usable [18]. In order to avoid a feature cascade, this section outlines two different feature sets: a minimum feature set and a feature set that users expect. The final section outlines graphical interface standards that should be applied to the GUI applications in order to increase ease of use [18].

1. Minimum Feature Set

A minimum feature set is the set of features necessary for the application to comply with an application's basic requirements. These features are required in order for the application to function as expected, yet they do not include features that are intended to increase usability (i.e., features users may expect). Since the configuration vector tool is an attempt to abstract away complexities, the first feature that meets the requirements is the main GUI application window. A GUI application ensures that a user does not need to know the syntactical structure of the underlying configuration vector file in order to create a syntactically correct file. Since the GUI portion of the application encompasses both the view and controller components, the GUI code incorporates the code for syntax and semantic error checking. The previous section described the main requirements for the configuration vector. Table 8 lists the minimum requirements for the application and the features of the application that meet these needs.

Requirement	Feature
Simplify configuration vector creation	GUI
Offline application that runs on standard operating systems	Java application
Read in configuration vector file (syntactically correct only)	Open dialog from the File menu of the application. A message is reported to the user if the input file is syntactically incorrect.
Write out configuration vector file	Save dialog from the File menu of the application. Allows the saving of a semantically incorrect file to a human-readable form only.
	Save As dialog from the File menu of the application (allows a copy of the vector to be saved). Allows the saving of a semantically incorrect file to a human-readable form only.
	Export dialog allows the user to write out a syntactically correct and semantically correct binary configuration vector file only.
Error checks	The syntax error check for an input file is accomplished immediately after attempting to open a file.
	The semantic error check is accomplished when the user attempts to export the current vector file. This check is also checked when the user presses the Check button.

Table 8. Minimum configuration vector tool features

2. Features Users Expect

Users expect features that increase the usability of an application. These features vary from application to application but include items that most users normally take for granted. A generic example is the ability to cut, copy, and paste text. Most users expect to find this feature as part of the application feature set. However, while this type of feature is a good addition to the overall list of features, it is not a feature necessary for the application to function. Thus, it is not a minimal feature. The list below outlines the features that a typical user would expect of the application.

- The ability to print a human-readable configuration vector without saving it to a file and opening it in another application
- Drag and drop a configuration vector file onto the application to open it
- Cut, copy, and paste for all the text fields inside the application
- Multiple instances of the application
- The ability to create user-defined presets
- The ability to load in default presets and user-defined presets
- Easy navigation between fields inside the application (i.e., tab goes to the next field, enter goes to the field below, etc.)

- The tables of the application are dynamic and only show the number of rows required for the current configuration vector instead of the maximum number of rows per table.
- The text color of permissions tables should be colored according to the type (e.g., RW colored red, RO colored black, NA colored green, WO colored blue)
- Sub-windows from the application (i.e., a process window or permissions window) should be the only window the user is able to focus on until the apply or cancel button on that window is clicked
- All primary functions of the application are assigned a keyboard shortcut
- All dialog windows should be the only window the user is able to focus on until the okay or cancel button for that dialog is clicked
- Text inside tables should be aligned appropriately based on the type of information contained in the specific column
- Error checking should be accomplished on the fly without having to press a check button
- Every field or button inside the GUI should have a tooltip that appears when the user hovers the mouse over the object for a period of time

The features users expect are niceties and not necessities for the application. However, these features add to the overall look and feel of the application. These features were incorporated into the application as time permitted. Many of these features were not implemented in the prototype and have been recommended for future development. The next section briefly outlines graphical standards that increase overall usability of the application.

3. Graphical Interface Standards

Because the configuration vector tool was created using the NetBeans IDE, it automatically supplied many of the graphical interface standards that should be applied to GUI applications. However, it is important to note the graphical standards followed [18]:

- Keyboard shortcuts adhere to the standard keyboard shortcuts used by modern day operating systems (i.e., Ctrl+S (windows) or Cmd+S (mac) for saving a file)
- System font is used for text in menus, dialogs, and full-sized controls
- Emphasized system font is used sparingly. The primary use is for the message text in text alerts and for titles of group settings boxes.

- All text input boxes use the application font as the default
- The font for labels is consistent across the application
- All sentences in the application are separated by a single space (a single space between the ending punctuation of one sentence and the first word of the next sentence).
- Labels for interface elements must be easy to understand and avoid technical jargon as much as possible
- All words in titles are capitalized except the following: articles (a, an, the), coordinating conjunctions (and, or), and prepositions of four or fewer letters except when the preposition is part of a verb phrase (e.g., Go To...).
- The ellipsis (...) character signifies that additional information is required before the operation can be performed.

This section outlined the requirements, features, and graphical standards that were used to design and implement the configuration vector tool GUI application. The next section shows the first conceptual designs of the application.

G. CONCEPTUAL DESIGN OF THE CONFIGURATION VECTOR TOOL

This section shows the initial concept design diagrams used for the configuration vector tool application. The graphics in this section are the result of six iterations of design. Throughout the iterations, many different aspects of the application were refined and improved in order to better meet the initial needs of the configuration vector tool.

The following set of figures show the finalized conceptual design sketch for the initial version of the configuration vector tool. Each figure is a translation from the configuration vector structs discussed in the Section B.

Figure 17 shows the partition table result after the translation. Figure 18 shows the datafile table. Figure 19 shows the memory table. Figure 20 shows the event counts table. Figure 21 shows the sequencers table. Figure 22 shows the partition-to-partition table. Figure 23 shows the subject resource table. Figure 24 shows the process window

that becomes visible to the user when the user clicks the processes column in the partition table. Figure 25 shows the permissions window that becomes visible to the user when a permissions column is clicked by the user.

The image shows a window titled "Configuration Vector Tool". At the top, there are several input fields: "Version: 2", "Magic: 2002", "TPA Partition: 0" (with a dropdown menu showing 1, 2, 3, 4...), "Number of Partitions: 1", "Number of Event Counts: 0", and "Number of Sequencers: 0". Below these fields is a table with tabs: "Partition", "Datafile", "Memory", "Event Counts", "Sequencers", "Partition-Partition Perms", and "Subject Resource Perms". The "Partition" tab is selected, showing a table with 8 rows (Partition 0 to Partition 7). Each row has columns for "#", "Identifier", "Description", "Time Slice %", "Max Memory", "Active", "# of Processes", and "Processes". The "Processes" column contains a "Click here..." button for each partition. At the bottom of the window are three buttons: "Check", "Export", and "Close".

#	Identifier	Description	Time Slice %	Max Memory	Active	# of Processes	Processes
Partition 0	101	Test-Kad	10	1024	Yes	1	Click here...
Partition 1	102	None	0	0	No	0	Click here...
Partition 2	103	None	0	0	No	0	Click here...
Partition 3	104	None	0	0	No	0	Click here...
Partition 4	105	None	0	0	No	0	Click here...
Partition 5	106	None	0	0	No	0	Click here...
Partition 6	107	None	0	0	No	0	Click here...
Partition 7	108	None	0	0	No	0	Click here...

Figure 17. Conceptual sketch of the partition table

Configuration Vector Tool

Version: TPA Partition: Number of Partitions:

Magic:

Number of Event Counts: Number of Sequencers:

Partition	Datafile	Memory	Event Counts	Sequencers	Partition-Partition Perms	Subject Resource Perms
#	Partition	Identifier	Privilege	Path	Permissions	
Datafile 0	3	201	1	/desktop/test.df	Click here...	
Datafile 1	0	202	0	none	Click here...	
Datafile 2	0	203	0	none	Click here...	
Datafile 3	0	204	0	none	Click here...	
Datafile 4	0	205	0	none	Click here...	
Datafile 5	0	206	0	none	Click here...	
Datafile 6	0	207	0	none	Click here...	
Datafile 8	0	208	0	none	Click here...	

Check Export Close

Figure 18. Conceptual sketch of the datafile table

Configuration Vector Tool

Version: TPA Partition: Number of Partitions:

Magic:

Number of Event Counts: Number of Sequencers:

Partition	Datafile	Memory	Event Counts	Sequencers	Partition-Partition Perms	Subject Resource Perms
#	Partition	Identifier	Privilege	Size	Permissions	
Memory 0	1	301	2	64	Click here...	
Memory 1	0	302	0	0	Click here...	
Memory 2	0	303	0	0	Click here...	
Memory 3	0	304	0	0	Click here...	
Memory 4	0	305	0	0	Click here...	
Memory 5	0	306	0	0	Click here...	
Memory 6	0	307	0	0	Click here...	
Memory 8	0	308	0	0	Click here...	

Check Export Close

Figure 19. Conceptual sketch of the memory table

Configuration Vector Tool

Version: TPA Partition: Number of Partitions:

Magic: Number of Event Counts:

Number of Sequencers:

Partition Datafile Memory Event Counts Sequencers Partition-Partition Perms Subject Resource Perms

#	Name	Privilege	Permissions
Event Count 0	Test EVCT	2	Click here...
Event Count 1	0	0	Click here...
Event Count 2	0	0	Click here...
Event Count 3	0	0	Click here...
Event Count 4	0	0	Click here...
Event Count 5	0	0	Click here...
Event Count 6	0	0	Click here...
Event Count 7	0	0	Click here...

Check Export Close

Figure 20. Conceptual sketch of the event counts table

Configuration Vector Tool

Version: TPA Partition: Number of Partitions:

Magic: Number of Sequencers:

Number of Sequencers:

Partition Datafile Memory Event Counts Sequencers Partition-Partition Perms Subject Resource Perms

#	Name	Privilege	Permissions
Sequencer 0	Test SEQ	2	Click here...
Sequencer 1	0	0	Click here...
Sequencer 2	0	0	Click here...
Sequencer 3	0	0	Click here...
Sequencer 4	0	0	Click here...
Sequencer 5	0	0	Click here...
Sequencer 6	0	0	Click here...
Sequencer 7	0	0	Click here...

Check Export Close

Figure 21. Conceptual sketch of the sequencers table

Subject Assignments

Processes

Identifier: 1 Number of Subjects: 1

Subjects

Privilege Level	Executable Path	Gate Path
0	/lpsk_kernel	/pl0.gts
1	None	None
2	None	None
3	None	

Check Apply Cancel

Figure 24. Conceptual sketch of the processes window and subjects table

Set Permissions

Object ## Permissions

Item Name	Permissions
Item Name 0	NA
Item Name 1	NA
Item Name 2	NA
Item Name 3	NA
Item Name 4	NA
Item Name 5	NA
Item Name 6	NA
Item Name 7	NA
Item Name 8	NA
Item Name 9	NA

Check Apply Cancel

Figure 25. Conceptual sketch of the permissions window and table

This section showed the transformation from the `lpsk.h` file to the conceptual graphical representation of a configuration vector. The interface shown in the figures of this section directly map to the actual implementation of the configuration tool described in the next section.

H. SUMMARY

This chapter began with the selection process used to select the correct programming language and associated development tools: Java Swing and the NetBeans IDE. Next, the chapter focused on the configuration vector format as described by the `lpsk.h` header file. The breakdown of the `lpsk.h` file not only helped the developer understand the configuration vector structure, but the model component of the MVC paradigm became apparent as did the primary graphical design element (i.e., the table). Next, the chapter described the table as the primary design element and then led the reader through the thought processes behind the table refinements in the tool. The next section focused strictly on the configuration vector requirements: reading a vector, writing a vector, and checking a vector. These requirements were then fully discussed in the features discussion of the configuration vector tool. These efforts led to the conceptual design of the configuration vector tool. The conceptual design was a series of sketches of what an actual implementation of the tool might look like. The next chapter discusses the implementation and testing of this conceptual design.

IV. IMPLEMENTATION AND TESTING

This chapter discusses the actual implementation and testing of the configuration vector tool. It is important to note that the tool that was implemented and tested, as described in this chapter, is a prototype. This chapter describes the underlying code written to generate both the delegate and model components by outlining the functionality of each of the Java class files. The chapter then describes the main functionality of the GUI. Then, the section shows screenshots of the prototype followed by a discussion of a concept of operation. The final section of the chapter presents the results of the tests used to validate the prototype. The tests were split into two general categories: error checking and input/output.

A. JAVA CLASS FILES

The Java class files that comprise the configuration vector tool are divided into four separate categories: command line tools, GUI components, model components, and additional controller components. The command line tool category contains a single Java class. The command line tool was used to check and verify the output of the configuration vector tool. The GUI component category contains the classes that create the GUI, check the data input for errors, and add data to the model components.

The model component category contains all components specific to the model. The model component files are special because the files are completely independent of any files in the other categories. All of these classes of the model component category are direct translations of the `lpsk.h` structs into Java-style representations. The `VectorStruct.java` file contains references to all of the other model component files. This means that in order to create a new and empty configuration vector, a developer only needs to instantiate a new `VectorStruct`.

The additional controller component category contains additional controller component files that were removed from the main GUI class file (i.e., CVToolGUI.java) to enable reuse across the entire application. This simplifies the code required by the main GUI class. Table 9 shows a detailed breakdown for each file of the configuration vector tool.

Java Classes		
Category	Name	Description
Command line tool	CVDump.java	This tool has the ability to read in a binary configuration vector file and generate a syntactically correct binary configuration vector file. This tool does not do any error checking. If the read-in configuration vector is incomplete, it will fill the blank fields with zeros upon writing the file out to disk. The primary purpose of this tool is to verify that the configuration vector generated by the configuration vector tool is correct. This is discussed in more detail in the testing section.
	CVToolGUI.java	This class is the Configuration Vector Tool GUI class. This class is the main class of the configuration vector tool (i.e., the view/controller component). This class creates the main window of the application and contains the application's main method. The class accesses all other classes in order to display the data to the user, allows the user to edit the data, and to save/export the data. The class also has the methods that accomplish the primary error checking of a given configuration vector.
GUI component	ProcSubjGUI.java	This class provides the code for the GUI window used to define a process. This also provides error checking for the subjects that it takes as input at its interface.
	PermsGUI.java	This class provides the code for the GUI window for declaring permissions.
Model component	VectorDefs.java	This class is the Java representation of the definitions section of the lpsk.h file.
	VectorStruct.java	This class is the Java representation of the vector_struct of the lpsk.h file.
	PartitionStruct.java	This class is the Java representation of the partition_struct of the lpsk.h file.
	DatafileStruct.java	This class is the Java representation of the

Java Classes		
Category	Name	Description
		datafile_struct of the lpsk.h file.
	MemoryStruct.java	This class is the Java representation of the memory_struct of the lpsk.h file.
	SynchronizationStruct.java	This class is the Java representation of the synchronization_struct of the lpsk.h file. This file is used to create the data structure for the eventcounts and sequences.
	PartPerm.java	This class is the Java representation of the part_perm[][] two-dimensional array of the lpsk.h file.
	SubjResPermStruct.java	This class is the Java representation of the subj_res_perms_struct of the lpsk.h file.
	ProcessStruct.java	This class is the Java representation of the process_struct of the lpsk.h file.
	SubjectStruct.java	This class is the Java representation of the subject_struct of the lpsk.h file.
Additional controller component	PrintVec.java	This class contains methods that allow the developer to print the configuration vector to the terminal screen for debugging and is also used by the user to create a human-readable configuration vector file.
	Utilities.java	This is a main controller class of the application. It sets the defaults for all tables and also assigns default values for every object in the configuration vector tool.
	Validator.java	The methods of this class are used to do error exception checking.

Table 9. Java class files of the configuration vector tool

B. PRIMARY GUI CLASS

Similar to the VectorStuct class, the CVToolGUI class is the primary Java class for the view/controller component. The CVToolGUI class contains the methods that create the view component (i.e., the main application window as well as the tabbed panel of tables). This class uses the two other classes in the GUI component category of Table 9 to gather additional data from the user (additional view components). The CVToolGUI class also contains the controller code that adds data from the view component to the model component. All data gathered by the view component is error checked by the controller component before it is sent to the model component data structures. The

following discussion explains the error checking functionality as well as the reading and writing of data from the CVToolGUI class (Table 10 provides a summary of this discussion).

The error checking for all the visible tables in the main window is accomplished in the CVToolGUI class. All of the error checking is accomplished when the user presses either the check button or the export button of the main application window. The tables embedded within the intermediate tables (i.e., the subject table contained in the process window of the partition table) are error checked when the user either presses the check button or the apply button of that window.

Live error checking within each table of the application (applicable to all windows) is also applied as the user enters data into each table. However, this error checking only restricts the type of information that may be entered into the specific cell of the table. For instance, the time slice column of the partition table will only allow integer values. If the user attempts to enter other data types, such as a string of characters, the user is unable to move on to another cell before fixing the data. No error message is displayed to the user in the initial prototype. Instead, the table cell border turns red to notify the user that there is a data type problem.

If all data entered in the CVToolGUI class passes the error checking methods of the class, the data is passed into the VectorStruct when the user presses the *export* button. The *export* button writes the binary configuration vector file to a user-specified location on disk. If the user wishes to bypass the error checking methods contained in the export button, the user must select the *save* or *save as* menu item from the file menu in the menu bar. *Save* writes the contents of the main application window as well as completed sub-windows (i.e., a process window or permissions window) to the same human-readable file that was originally opened by the user. If the user opened a binary file, the *save* command does not save the file in the binary format. In this case, the *save* command acts like a *save as* command and writes a human-readable file to a user-specified location. The *save as* command enables the user to save a human-readable file to a user-specified location with a user-specified file name.

It is important to notice the distinction between the *save* and *save as* commands and the *export* command. The *save* and *save as* commands will only write a human-readable file to disk. The *export* command creates the binary file that is used to initiate the LPSK platform. These choices provide the user with a great deal of flexibility. It allows the user to create or open a vector file, save the file to disk, exit the application, and open the file later without having to ensure the vector file is error-free or complete. In addition, this environment prevents the user from creating an invalid binary file. This saves the user from attempting to boot a LPSK platform with a syntactically or semantically invalid binary configuration vector, which will only result in a halt of the platform.

The final major requirement of the application is to read a previously created vector file. As previously discussed, the application will only open syntactically correct vector files. However, similar to the *save* and *save as* command, the configuration vector tool will allow a vector that has incorrect semantic values to be opened. This allows the user the ability to correct any invalid data before exporting a binary file. As stated in the previous section, the tool will not allow the user to export an invalid binary file even if the file opened by the user has invalid values.

Command	Action
New	Creates a blank configuration vector file by instantiating a new VectorStruct.
Save	Allows the user to write data from the GUI to a human-readable file only. The file is written to the same human-readable file that was originally opened by the user. If a binary file was opened, the <i>save</i> command defaults to the <i>save as</i> command. The data written to a file may be semantically incorrect but will be syntactically correct.
Save As	Similar to the <i>save</i> command except it allows the user to specify the location and file name of the human-readable file to write to disk.
Check	Applies all semantic error checks, notifying the user of any errors encountered.
Export	Accomplishes the same tasks as the <i>check</i> command but also writes a binary configuration vector file to a user-specified location and file name if and only if all of the error checks were passed.
Apply	Used in the sub-windows and applies all error checks, notifying the user of any errors encountered.

Table 10. List of the basic commands of the configuration vector tool

This section described the functionality of the primary CVToolGUI class. The next section discusses the features not implemented in this prototype.

C. PROTOTYPE

This section details the implementation of the configuration vector tool prototype. The section is broken into two subsections containing screenshots of all tables of the application and detailed explanations of all items of the application.

1. Screenshots

This section shows the actual implementation screenshots of the configuration vector tool. Each figure in this section corresponds directly to the conceptual designs discussed earlier. Figure 26 shows the implementation of the partition table. Figure 27 shows the implementation of the datafile table. Figure 28 shows the implementation of the memory table. Figure 29 shows the implementation of the eventcounts table. Figure 30 shows the implementation of the sequencers table. Figure 31 shows the implementation of the partition-to-partition table. Figure 32 shows the implementation of the subject-resource permissions table. Figure 33 shows the implementation of the process window and associated subject table. Figure 34 shows an example of a specific implementation of a permissions window.

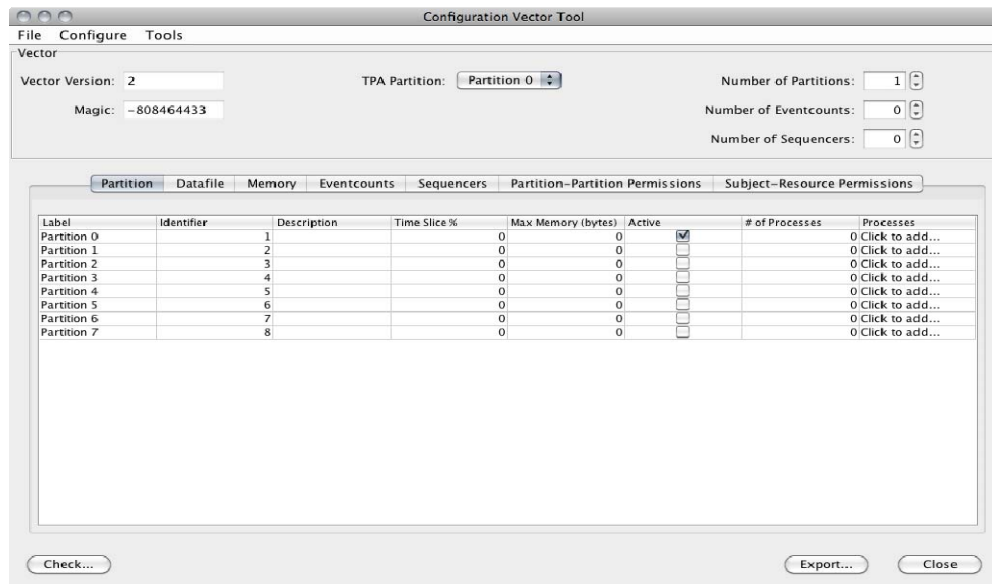


Figure 26. Partition table view of the application

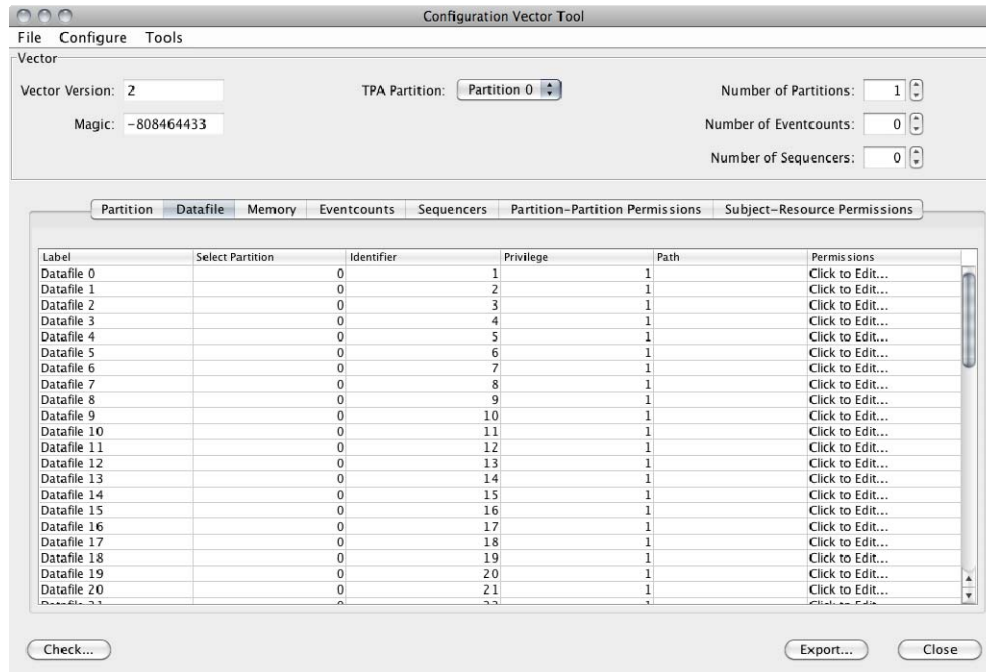


Figure 27. Datafile table view of the application

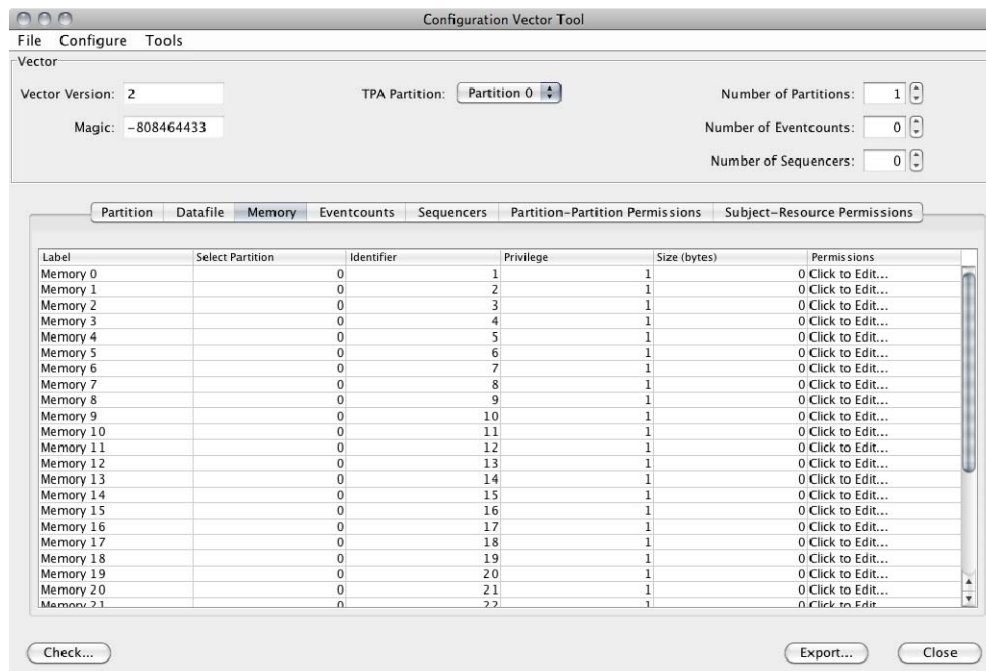


Figure 28. Memory table view of the application

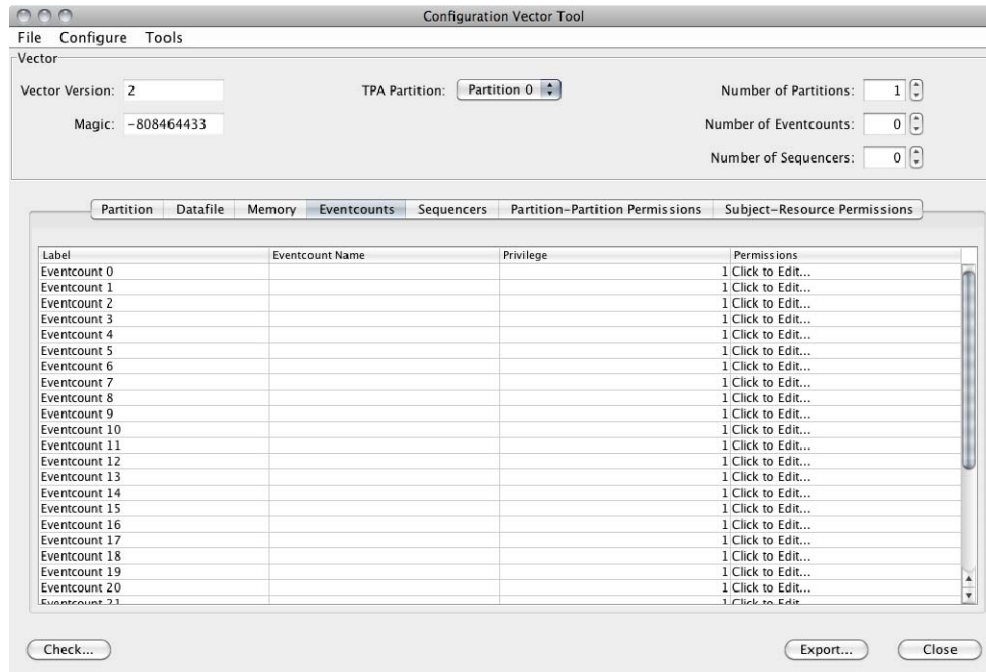


Figure 29. Eventcounts table view of the application

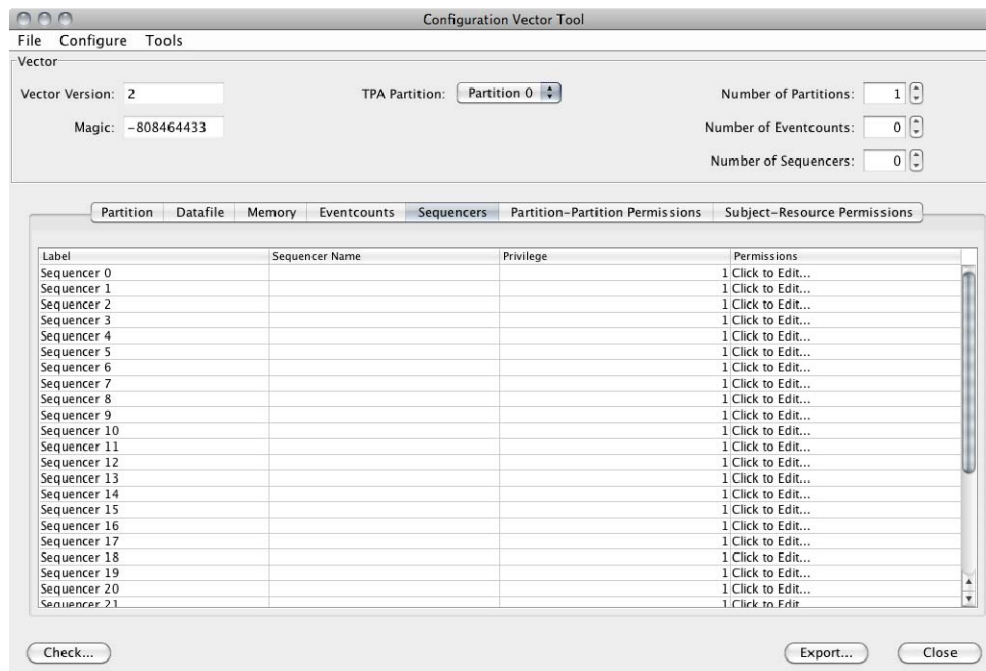


Figure 30. Sequencers table view of the application

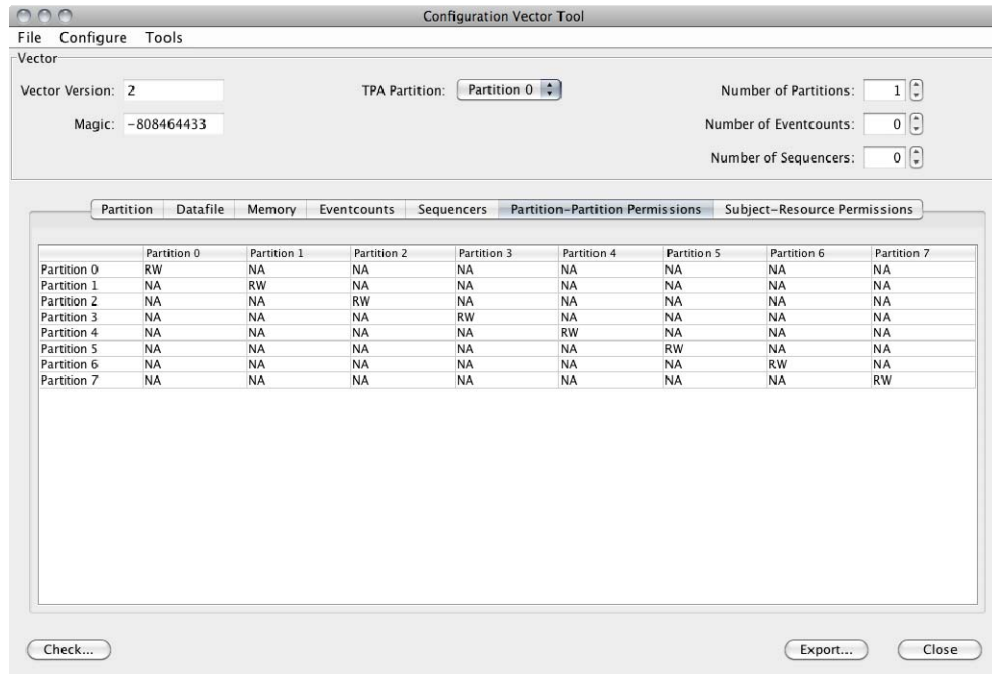


Figure 31. Partition-to-partition table view of the application

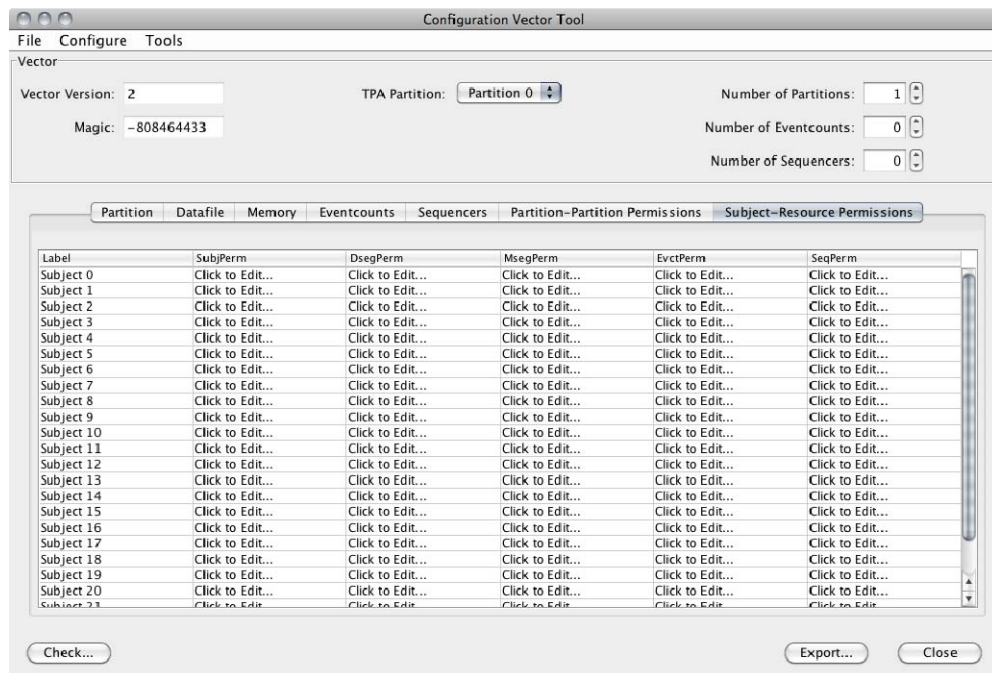


Figure 32. Subject-resource permissions table view of the application

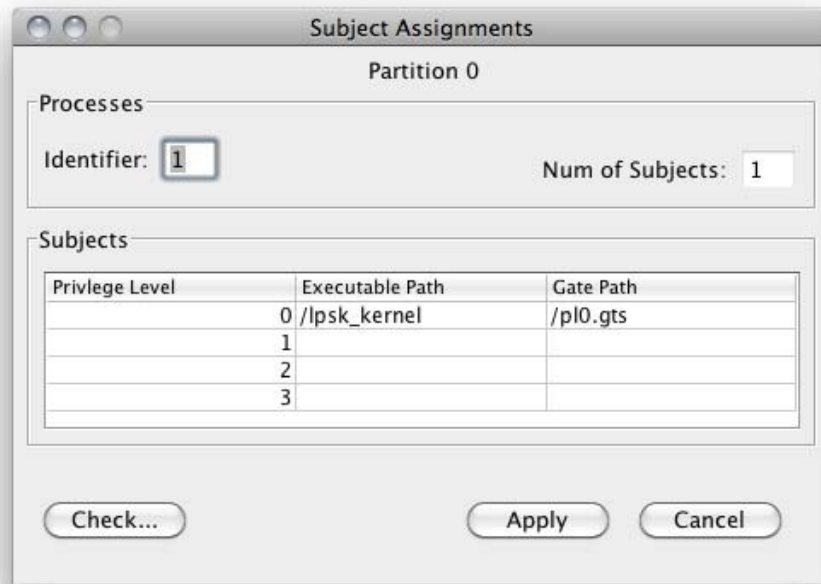


Figure 33. Process and subject window of the application

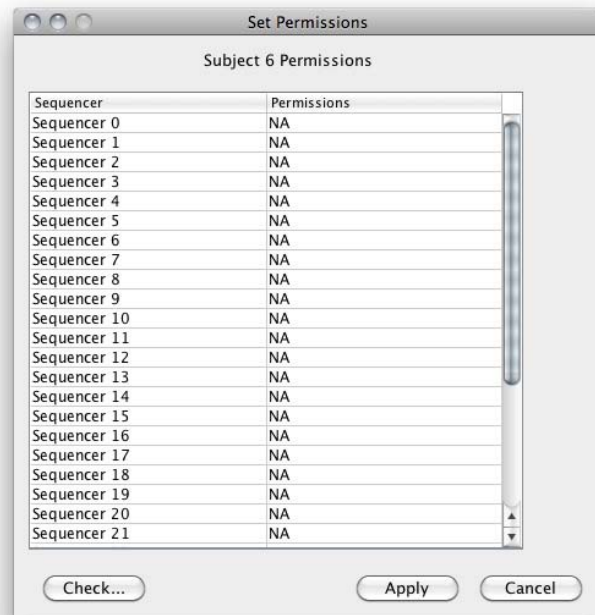


Figure 34. Permissions window and associated table view of the application

2. Concept of Operation

The purpose of this section is to explain how the configuration vector tool operates. This section can be treated as a brief tutorial on how to create, edit, save, and export a configuration vector using the tool. The section first describes each window presented to the user and then provides basic steps for accomplishing typical tasks.

After launching the configuration vector tool, the user is presented with the main application window (see Figure 26). The main application window is broken into two distinct areas. The top third of the window contains the vector attributes panel and the bottom two-thirds contains the tabbed tables panel. The vector attributes panel includes the six items discussed during the design phase of this research. The *Version* and *Magic* fields cannot be changed by the user and are fixed by the configuration vector tool (or the values of these fields are set upon opening a previously created configuration vector). In a future prototype, the version field will be updatable by the user through a *preferences* window (see Chapter V, Section B). The far right side of the *vector attributes* panel allows the user to specify the number of partitions, eventcounts, or sequencers for the configuration vector (see Figure 35). Because a configuration vector must have at least one active partition, the default value for the *Number of Partitions* is one. The *TPA partition* is a dropdown menu that includes the maximum number of partitions available. The user must select the desired partition to set as the TPA partition (only one partition may be set as the TPA partition).



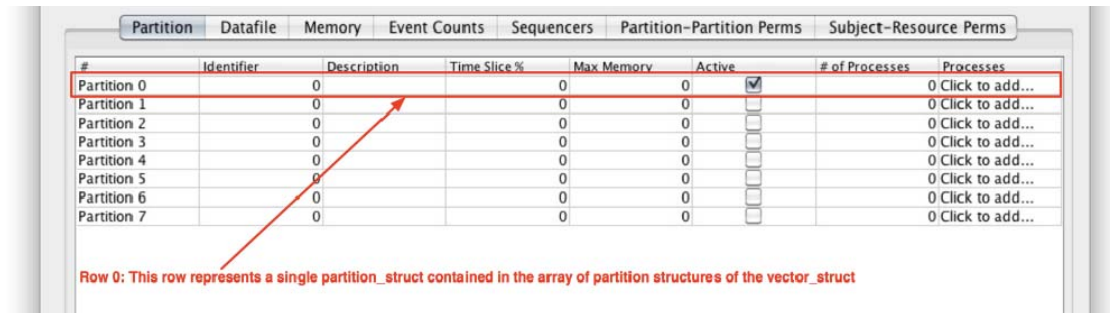
The screenshot shows a panel titled "Vector" with the following controls:

- Vector Version:** A text field containing the value "2".
- Magic:** A text field containing the value "-808464433".
- TPA Partition:** A dropdown menu currently showing "Partition 0".
- Number of Partitions:** A numeric spinner control set to "1".
- Number of Eventcounts:** A numeric spinner control set to "0".
- Number of Sequencers:** A numeric spinner control set to "0".

Figure 35. The vector attribute panel of the main window

As stated in the design discussion, each table of the tabbed table panel represents a specific configuration vector structure. A single row of any table represents an individual item of a specific structure. For instance, in the partitions table, row zero

represents the specific partition information for Partition 0 (see Figure 36). All tables within the configuration vector tool have this same relationship between the GUI representation and the underlying data structure.



#	Identifier	Description	Time Slice %	Max Memory	Active	# of Processes	Processes
Partition 0	0		0	0	<input checked="" type="checkbox"/>	0	Click to add...
Partition 1	0		0	0	<input type="checkbox"/>	0	Click to add...
Partition 2	0		0	0	<input type="checkbox"/>	0	Click to add...
Partition 3	0		0	0	<input type="checkbox"/>	0	Click to add...
Partition 4	0		0	0	<input type="checkbox"/>	0	Click to add...
Partition 5	0		0	0	<input type="checkbox"/>	0	Click to add...
Partition 6	0		0	0	<input type="checkbox"/>	0	Click to add...
Partition 7	0		0	0	<input type="checkbox"/>	0	Click to add...

Row 0: This row represents a single partition_struct contained in the array of partition structures of the vector_struct

Figure 36. View of row zero of the partition table

The next step after opening the tool is to begin filling in data. The user should fill out all tables and fields with the desired data. This includes adding processes to a specific partition or setting the permissions for the eventcounts, sequencers, and subject-resource permissions. As previously stated, data types are validated on the fly. This means a user attempting to enter a letter in the time slice or maximum memory column of the partition table will not be able to exit the cell until the correct data type is entered. The table does not check the bounds on the fly. Once data is entered in the tables of the tool, the user can do one of three things:

1. check the vector for errors by pressing the check button,
2. save the current vector in a human-readable format by selecting the *save* or *save as* menu item from the file menu, or
3. attempt to export a binary configuration vector file to a specific location.

If the user presses the *check* button, the tool will error check all cells in all the tables and report any errors to the user with a popup dialog that provides the specific error and recommendation for fixing the error. The tool will also reset the value of the invalid cell to the default value. No file is saved or exported by clicking the check button.

If the user selects the *save as* menu item, the current values in the configuration vector tool are written to a human-readable file. Although this file is syntactically correct, the values within the file may not be semantically valid. Saving a vector file

allows the user to continue editing the file later. The only difference between the *save* and *save as* command is that the *save as* command allows the user to select the name and location of the human-readable file.

If the user presses the export button, the vector is checked just as if the *check* button was pressed. The export will fail if the check encounters an error. The error is reported to the user in the same manner as clicking the *check* button. If the check is passed, the tool provides the user a dialog box that allows the user to select the name and location of the output file. Once this is completed a binary vector is exported to the desired location.

The final main feature of the tool is opening a configuration vector file and creating a new vector file. The tool can open either a syntactically correct human-readable file or a syntactically correct binary file. If the file is invalid, the tool reports an error message with the location of the first error encountered to the user. Otherwise, the file is opened, the data is read into the tool and the appropriate fields are filled. The user opens a configuration vector file by selecting the *open* menu item from the file menu in the menu bar. A dialog window is presented to the user allowing the user to locate and select the desired file. Once the vector file is opened, the values may be edited and saved or exported as desired. Creating a new configuration vector simply requires the user to start the configuration vector tool or select the *new* menu item from the file menu in the menu bar.

This section outlined the main features of the configuration vector tool and provided a brief tutorial on how a user might use the configuration vector tool. The final section in this chapter describes the testing procedures that were performed against the tool and the results of those tests.

D. TESTING

This section outlines the test plan and procedures used to validate the configuration vector tool. Testing the configuration vector occurred in two phases. The first phase tested the bounds of the data within the fields while in the tool. The second

phase tested the input/output capabilities of the tool. The subsequent sections report the testing plan and results of each testing phase.

1. Phase I: Error Checking

This phase tested the bounds error checking ability of the configuration vector tool. All input fields of the configuration vector tool were checked to ensure that each complied with the respective bounds listed in the requirements section. The tool enforces the bounds by restricting the user's ability to input incorrect data in addition to checking the data when the user presses the *check*, *apply*, or *export* button. The following tables, Table 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20 show the GUI mechanism used to enforce a restriction and the restriction that is enforced. Each of the fields listed in the tables were checked using the following procedures:

- Launch the configuration vector tool
- Verify every table to ensure that all values in each field are set to the default value
- Enter edge-case values in an individual field (e.g., if the field has a bound, enter data that tests that bound). Ensure the value is set to the default value before moving on to the next tested field.

Vector Attributes Panel		
Field Name	Restriction Mechanism	Enforced Restriction
Version	Read-only text field	Read-only
Magic	Read-only text field	Read-only
TPA Partition	Dropdown selection box	Only one value may be selected by the user (up to the maximum number of partitions)
Number of Partitions	Spinner text field	Only values greater than 0 and less than or equal to the maximum number of partitions may be entered
Number of Eventcounts	Spinner text field	Only values greater than or equal to 0 and less than the maximum number of eventcounts may be entered
Number of Sequencers	Spinner text field	Only values greater than or equal to 0 and less than the maximum number of sequencers may be entered

Table 11. Vector attributes panel restrictions

Partition Table		
Field Name	Restriction Mechanism	Enforced Restriction
Identifier	Integer only cell	Ensures only integers may be entered
	Check/Export button	Ensures the identifier is unique to the partition table and greater than zero
Description	Check/Export button	Ensures the length of the text entered is less than the maximum description length defined in the VectorDefs.java file
Time Slice %	Integer only cell	Ensures only integers may be entered
	Check/Export button	Ensures individual time slice values are between 0 and 100 inclusive
		*Set to 0 if the partition is not active (dependent on the active field of the partition table) *Ensures the sum of all time slices across all defined partitions equals 100
Max Memory	Integer only cell	Ensures only integers may be entered
	Check/Export button	Ensures the maximum memory is greater than zero
Active	Checkbox	Ensures that only on or off can be checked
	Check/Export button	*Ensures that at least one partition of the defined partitions is set to active and that partition has at least one process
# of Processes	Dropdown selection box	Ensures only values between 0 and the maximum number of processes may be selected
	Check/Export button	*Ensures that if the active box is unchecked, then the number of processes is set to zero
Processes	Read-only & click to launch processes window	n/a
	Check/Export button	*Ensures that if the processes is set to zero then the subjects defined in the processes window are cleared.

* Requires more complex test procedures than discussed above

Table 12. Partition table restrictions

Datafile Table		
Field Name	Restriction Mechanism	Enforced Restriction
Partition	Dropdown selection box	Ensures that the user may only select a partition number from the defined partitions
Identifier	Integer only cell	Ensures only integers may be entered
	Check/Export button	Ensures the identifier is unique to the partition table and greater than zero
Privilege	Dropdown selection box	Only allows the selection of 0, 1, 2, or 3
Path	Check/Export button	Ensures the length of the text entered is less than the maximum path length defined in the VectorDefs.java file
Permissions	Read-only & click to launch permissions window	n/a

Table 13. Datafile table restrictions

Memory Table		
Field Name	Restriction Mechanism	Enforced Restriction
Partition	Dropdown selection box	Ensures that the user may only select a partition number from the defined partitions
Identifier	Integer only cell	Ensures only integers may be entered
	Check/Export button	Ensures the identifier is unique to the partition table and greater than zero
Privilege	Dropdown selection box	Only allows the selection of 0, 1, 2, or 3
Size	Integer only cell	Ensures only integers may be entered
	Check/Export button	Ensures that the maximum size does not exceed the maximum physical memory of the system
Permissions	Read-only & click to launch permissions window	n/a

Table 14. Memory table restrictions

Eventcounts Table		
Field Name	Restriction Mechanism	Enforced Restriction
Name	Check/Export button	Ensures the length of the text entered is less than the maximum path length defined in the VectorDefs.java file
Privilege	Dropdown selection box	Only allows the selection of 0, 1, 2, or 3
Permissions	Read-only & click to launch permissions window	n/a

Table 15. Eventcounts table restrictions

Sequencers Table		
Field Name	Restriction Mechanism	Enforced Restriction
Name	Check/Export button	Ensures the length of the text entered is less than the maximum path length defined in the VectorDefs.java file
Privilege	Dropdown selection box	Only allows the selection of 0, 1, 2, or 3
Permissions	Read-only & click to launch permissions window	n/a

Table 16. Sequencer table restrictions

Partition-to-Partition Perms Table		
Field Name	Restriction Mechanism	Enforced Restriction
All cells	Dropdown selection box	Only one value may be selected: NA, RO, RW, WO

Table 17. Partition-to-partition table restrictions

Subject-Resource Perms Table		
Field Name	Restriction Mechanism	Enforced Restriction
All Cells	Read-only & click to launch permissions window	n/a

Table 18. Subject-resource table restrictions

Process Window		
Field Name	Restriction Mechanism	Enforced Restriction
Identifier	Read-only field	Value is unique to across all processes windows and greater than or equal to zero
Number of Subjects	Read-only field	n/a
	Check/Apply button	Calculated by counting the number of subjects defined in the executable path field of the subjects table.
Executable Path	n/a	n/a
	Check/Apply button	Ensures the length of the text entered is less than the maximum path length defined in the VectorDefs.java file
Gate Path	n/a	n/a
	Check/Apply Button	Ensures the length of the text entered is less than the maximum path length defined in the VectorDefs.java file Ensures the PL3 gate is always empty

Table 19. Process window restrictions

Permissions Window		
Field Name	Restriction Mechanism	Enforced Restriction
Permissions	Dropdown selection box	Only one value may be selected: NA, RO, RW, WO (Permissions originating from the sequencers table may only be NA and RO)

Table 20. Permissions window restrictions

Every field in the tool was verified to work as described by the restrictions listed in the tables above except for the starred restrictions. These fields required a more complicated test procedure. For each starred item, the tests were conducted by first entering correct data into the tables. This means that all data initially entered into the tool was verified as correct before beginning a test. The information then was modified to test each error case. The starred items are represented by the Special Tests (SPT) in the list below:

- SPT1 (Partition table – Time slice %): the *check/export* button ensures that the time slice is set to 0 if the partition is not active (dependent on the active field of the partition table)
- SPT2 (Partition table – Time slice %): the *check/export* button ensures the sum of all time slices across all defined partitions equals 100.
- SPT3 (Partition table – Active): the *check/export* button ensures that at least one partition of the defined partitions is set to active.
- SPT4 (Partition table – # of processes): the *check/export* button ensures that if the active box is unchecked, then the number of processes field is set to zero and all subjects are cleared from that process.
- SPT5 (Partition table – Processes): the *check/export* button ensures that if the processes is set to zero then the subjects defined in the processes window are cleared.

The test procedure for these starred items is described in Table 21. For these tests, the configuration vector tool was launched and default values were verified across all tables. After entering the specific test data described in Table 21, the check/export button was pressed to activate the checking mechanisms. Finally, after confirming the expected results, the configuration vector tool was closed and then the next set of tests was started.

Test Name	Test Procedure	Expected	Observed
SPT1	Set the number of partitions to 2. Set TPA Partition to Partition 0. For Partition 0: - set the time slice cell to 1. - uncheck the active checkbox. For Partition 1: - set time slice cell to 99. - check the active checkbox. Press the check/export button.	- Error reported to user. - Partition 0 is cleared of all data. - The number of partitions field is corrected automatically. - A time slice error is presented to the user. All time slice %s across all active partitions must be equal to 100.	Same as expected
	Set the number of partitions to 2. Set TPA Partition to Partition 0. For Partition 0: - set the time slice cell to 1. - check the active checkbox. For Partition 1: - set time slice cell to 99. - check the active checkbox. Press the check/export button.	No error reported.	Same as expected
SPT2	Set the number of partitions to 2. Set TPA Partition to Partition 0. For Partition 0: - set the time slice cell to 3. - check the active checkbox.	- Error reported to the user - Time slice % of defined partitions exceeds 100. Please correct time slice % across all defined partitions.	Same as expected.

Test Name	Test Procedure	Expected	Observed
	For Partition 1: - set time slice cell to 98. - check the active checkbox. Press the check/export button.		
	Set the number of partitions to 2. Set TPA Partition to Partition 0. For Partition 0: - set the time slice cell to 1. - check the active checkbox. For Partition 1: - set time slice cell to 98. - check the active checkbox. Press the check/export button.	- Error reported to the user. - Time slice % of defined partitions does not equal 100. Please correct time slice % across all defined partitions.	Same as expected.
	Set the number of partitions to 2. Set TPA Partition to Partition 0. For Partition 0: - set the time slice cell to 2. - check the active checkbox. For Partition 1: - set time slice cell to 98. - check the active checkbox. Press the check/export button.	No error reported.	Same as expected.
SPT3	Load two complete partitions into the table. Set the number of partitions to 2. Set TPA Partition to Partition 0. Uncheck all active checkboxes. Press the check/export button.	- Error reported to the user. - At least one partition must be active.	Same as expected.
	Load two complete partitions into the table. Set the number of partitions to 2. Set TPA Partition to Partition 0. Uncheck all active checkboxes. For Partition 0: - set the time slice cell to 100. - check the active checkbox. Press the check/export button.	- Error reported to the user. - User is presented with an option to clear additional information in the table and automatically correct the number of partitions to the value 1.	Same as expected.
SPT4	Set the number of partitions to 2. Uncheck all active checkboxes. For Partition 0: - set the time slice cell to 100. - check the active checkbox. For Partition 1: - set # of processes to 1.	- Error reported to the user. - User is presented with an option to clear additional information in the table and automatically correct the number of partitions to the value 1.	Same as expected.
	Set the number of partitions to 2. Uncheck all active checkboxes. For Partition 0: - set the time slice cell to 100. - check the active checkbox.	No error reported.	Same as expected.
SPT5	Set the number of partitions to 2. For Partition 0:	- Error reported to the user. - User is informed that the partition	Same as expected.

Test Name	Test Procedure	Expected	Observed
	<ul style="list-style-type: none"> - set all fields with correct values. - set the time slice cell to 50. - check the active checkbox. - set number of processes to 1. - ensure that there are subjects. <p>For Partition 1:</p> <ul style="list-style-type: none"> - set all fields with correct values. - set the time slice cell to 50. - check the active checkbox. - set number of processes to 0. - ensure that there are subjects. 	does not have subjects.	

Table 21. Special tests table for starred entries

2. Phase II: Input/Output

This phase tested the tool's ability to perform input and output operations to disk. The main functions tested were creating a new vector, opening a vector, saving a vector (to include *save* and *save as* commands), and exporting a binary vector. Since all files created by the tool will be either human-readable or binary, it is important to discuss how it was determined if the files written to disk were correct and valid. For a human-readable file, manually verifying the content in the generated file was sufficient (although tedious). The binary file, however, required verification that is more complex. For this reason, the CVDump command line tool was developed and used.

The CVDump tool is a command line tool. It reads in a binary vector file and writes the same vector file to disk with a different name (appends *_write* to the original file name) using the same methods employed by the GUI tool. This allows the two files (original and generated) to be compared against one another. Before CVDump could be used for verifying vector files generated by the tool, CVDump itself was verified to work correctly. This was accomplished by obtaining a known valid and correct binary configuration vector file (this vector was created by hand and used to successfully initialize the LPSK to a secure state). The known binary file was read by the CVDump tool. The CVDump tool then created a new binary file based on this known binary file. The two files sizes and hashes were then compared. If both the size and hash of the two files were the same, then the CVDump tool generated a correct and valid binary

configuration vector file. The known good binary configuration vector file obtained for this test was vect_out. The commands used to generate the outputs are listed in Table 22 and results of the comparison are shown in Table 23.

Description	Command	Results
1. Obtain a specific file size	<code>ls -l cvt/filename*</code>	File size of the desired file is printed on the screen.
2. Hash a specific file	<code>md5 cvt/filename*</code>	The MD5 hash of the desired file is printed on the screen
3. Execute CVDump	<code>java cvt/CVDump -d -v cvt/filename</code>	Executes the CVDump tool that creates a new binary file: filename_write
4. Hash the new file	<code>md5 cvt/filename_write</code>	The MD5 hash of the desired file is printed on the screen

Table 22. Commands used to verify the CVDump tool

Version	File Name	Size	MD5 Hash
Original	vect_out	35320	b9cd53d8d502be0a2482f3acdd0b358c
Generated	vect_out_write	35320	b9cd53d8d502be0a2482f3acdd0b358c

Table 23. Verification of CVDump command line tool

The verification of the CVDump tool made it possible to use the tool to check the output binary files generated by the configuration vector tool. As long as a binary vector file generated by the graphical tool and then processed through the CVDump as described above hashes to the same value, it was assumed that the graphical tool generated a correct and valid binary configuration vector file. With the CVDump tool verified, the remainder of the section outlines the test procedures and results of those tests.

The test plan for checking the values of the input fields is as follows (IO represents Input/Output Test):

- IO1: Create a new configuration vector by opening the tool.
- IO2: Create a new configuration vector by selecting the *new* menu item from the file menu in the menu bar.
- IO3: Attempt to open a valid and an invalid existing human-readable configuration vector file by selecting the *open* menu item from the file menu in the menu bar.

- IO4: Attempt to open a valid and an invalid existing binary configuration vector file by selecting the *open* menu item from the file menu in the menu bar.
- IO5: Attempt to save an opened human-readable configuration vector by selecting the save menu item from the file menu in the menu bar.
- IO6: Attempt to save an opened binary configuration vector by selecting the *save* menu item from the file menu in the menu bar.
- IO7: Attempt to save an opened human-readable configuration vector by selecting the *save as* menu item from the file menu in the menu bar.
- IO8: Attempt to save the opened binary configuration vector by selecting the *save as* menu item from the file menu in the menu bar.
- IO9: Attempt to export a valid configuration vector by selecting the *export* menu item from the file menu in the menu bar or pressing the *export* button.
- IO10: Attempt to export an invalid configuration vector by selecting the *export* menu item from the file menu in the menu bar or pressing the *export* button.
- Capture the results of these tests in a table.

The test results were captured in several tables. These tables were divided based on the attempted function performed (i.e., new, open, save/save as, and export). Table 24 captures the test results after executing the *new* command. Table 25 captures the test results after executing the *open* command. Table 26 captures the test results after executing the *save* or *save as* commands. Table 27 captures the test results after executing the *export* command. All tables contain the name of the test (e.g. IO1 as listed above), the procedure used, the expected result, and the observed results. For all binary file comparisons, CVDump was used and the resulting file sizes and hashes were compared.

New			
Name	Procedure	Expected	Observed
IO1	- Launch the application	All fields of the tools are set to defaults	Same as expected
IO2	- Launch the application - Add data to fields - Execute File > New	A message asking the user if he is sure he wishes to discard changes and create a new vector	Same as expected

Table 24. Test results for creating a new configuration vector

Open			
Name	Procedure	Expected	Observed
IO3	<i>Human-readable</i> - Launch the application - Execute File > Open - Select valid test file	Application fills all fields correctly	Same as expected
	<i>Human-readable</i> - Launch the application - Execute File > Open - Select invalid test file	Application fails to open the file and reports an error message to the user	Same as expected
IO4	<i>Binary</i> - Launch the application - Execute File > Open - Select valid test file	Application fills all fields correctly	Same as expected
	<i>Binary</i> - Launch the application - Execute File > Open - Select invalid test file	Application fails to open the file and reports an error message to the user	Same as expected

Table 25. Test results for opening a configuration vector

Save/Save As			
Name	Procedure	Expected	Observed
IO5	<i>Human-readable</i> - Launch the application - Execute File > Open - Select valid test file - Change a value - Execute File > Save - Close the application - Open saved file in a text editor and look for the change	The original human-readable file should contain the change added by the application and should be viewable in a text editor.	Same as expected
IO6	<i>Binary</i> - Launch the application - Execute File > Open - Select valid test file - Change a value - Execute File > Save - Close the application - Open saved file in a text editor and look for the change	The application should ask the user to specify the name and location of the file to save. The change should be viewable in a text editor.	Same as expected
IO7	<i>Human-readable</i> - Launch the application - Execute File > Open - Select valid test file - Change a value - Execute File > Save As - Close the application - Open saved file in a text editor and look for the change	The same as IO5 with the addition that the application should ask the user to specify the name and location of the file to save.	Same as expected

	change		
IO8	<i>Binary</i> - Launch the application - Execute File > Open - Select valid test file - Change a value - Execute File > Save As - Close the application - Open saved file in a text editor and look for the change	The same as IO6	Same as expected

Table 26. Test results for saving a configuration vector

Export			
Name	Procedure	Expected	Observed
IO9	<i>Binary</i> - Launch the application - Execute File > Open - Select valid test file - Execute Export Button - Close the application - Use CVDump to verify binary file.	The application should successfully write a binary file to disk. Using the CVDump tool, the hashes of the two files should match	Same as expected
IO10	<i>Binary</i> - Launch the application - Execute File > Open - Select valid test file - Change a value so that the vector is now invalid - Execute Export Button	The application should identify the error and not complete the export	Same as expected

Table 27. Test results for exporting a configuration vector

3. Test Summary

The tests completed in this section tested the configuration vector tool in two phases. The first phase tested the bounds of all fields inside the configuration vector tool. The tests of that phase were completed upon entry of data into the tool and when the *check* or *export* button was pressed. The second phase tested the input/output capability of the tool. The second phase was completed using the CVDump command line tool in combination with MD5 hashing. Together, these two phases provide a comprehensive test of the basic operations of the configuration vector tool prototype.

E. SUMMARY

This chapter discussed the prototype implementation of the conceptual design outlined in Chapter III. The chapter began with a description of the Java files used to implement the prototype. The next section showed screenshots of the implementation along with the concept of operations. The final section in this chapter outlined the two sets of tests (i.e., error checking and input/output) used to validate the initial prototype. The next chapter discusses the results, problems encountered, and recommendations for future work on the configuration vector tool.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

The initial implementation of the conceptual design of the configuration vector tool, as described in this thesis, is the first in a possible series of prototypes. In creating this prototype, this research has demonstrated that it is possible to build a GUI for creating configuration vectors. The prototype is successfully able to read and write valid binary configuration vectors. A valid configuration vector is one where all fields pass a bounds check as well as a semantic check. The prototype can also read and write human-readable configuration vectors. The human-readable files only pass a bounds check allowing a user to save a configuration vector that may be incomplete. This allows the user to complete a configuration vector at a later time. However, the prototype still requires substantial work before it should be considered to be a fully functional product. The four main reasons for its incompleteness are discussed in this chapter and are summarized in the conclusion. This chapter starts with a discussion of two significant problems encountered by the developer, followed by a discussion of incomplete features. Finally, the chapter ends with a discussion of suggestions for future work and the conclusion.

A. PROBLEMS ENCOUNTERED

During the development of the configuration vector tool prototype, two major problems were encountered. These hindered the development of more user-friendly features that were discussed in Chapter III, Section F. The following two sections discuss these problems in detail.

1. wxPython

The first problem encountered during development was the initial choice of wxPython as the preferred language. Although wxPython provides a complete feature set for the creation of GUIs, the developer's inexperience with wxPython was considerable. wxPython provides no support for easy placement of GUI elements in desired locations on an interface canvas. Initially, the interface was designed completely by hand without the use of a GUI builder. This proved to be more time-consuming and difficult than

expected. Thus, the search for a GUI builder application was started. During this search, the Boa Constructor IDE (see Chapter III, Section A) was discovered.

The developer found the Boa Constructor IDE unpolished and cumbersome. Since the majority of the work to this point was completed in wxPython, the developer was reluctant to change programming languages. This reluctance resulted in a considerable loss of time. Since Boa Constructor was not found to be user-friendly, a search for another GUI builder began. Unfortunately, the other Python-based GUI builders did not provide any additional help. Thus, the choice was made to move to Java and use the NetBeans IDE for development. Development went quickly after this choice was made.

2. NetBeans Tables

The NetBeans IDE Swing GUI builder made the creation of simple GUIs quite easy. However, it was not without its problems. For the most part, the objects of the GUI can be graphically placed on a canvas. Unfortunately, the ability to highly customize the graphical objects was not as simple. Specifically, customizing the table object was quite difficult.

In order to customize a table object in NetBeans, a developer must go through the NetBeans table builder interface. Unfortunately, few customization options are presented to the developer. Thus, creating a customized table that can be displayed in complex ways (e.g., only showing a certain number of rows) is difficult. Because of this difficulty in table customization, the tables in the prototype configuration vector tool were kept as simple as possible. Thus, customized features, such as controlling the number of rows visible to the user or more elegant error controls, were not implemented. The next section discusses a possible solution to the table customization problem encountered in NetBeans as well as additional future work.

B. INCOMPLETE FEATURES

The initial prototype of the configuration vector tool meets all of the basic requirements by implementing the basic feature set. This prototype of the tool has

several issues with the basic feature set that must be mentioned. Before discussing these issues, it is important to note that this release of the tool does not address any of the features users expect (see Chapter III, Section F).

The tool lacks complete and elegant error checking. The tool is able to verify and check the bounds on all attributes in the tables as well as provide dependency error checking between cells of the tables. The tool currently checks that the values for the attributes are within the defined limits when either the check button is pressed or when the export button is pressed. A more elegant solution is to check the bounds as the user enters each value and not allow the user to change from one cell to another without correcting the identified error. Dependency errors should still be checked when the user clicks a check button or attempts to export a vector.

Another incomplete feature is associated with the way the tool displays the tables to the user. The tool only displays the maximum number of attributes available to the user. The maximum number of attributes of each table is displayed regardless of whether or not the user requires all of the attributes. For instance, according to the MAX_PARTITIONS value stored in the VectorDefs.java file, the maximum number of partitions is eight. Thus, the configuration vector tool displays eight partitions in the partition table. However, in the vector attributes panel, there is an attribute to specify the number of partitions defined in the vector. Thus, creating a table that dynamically changes as the number of partitions field changes is desirable. For this prototype, changing the number of partitions does not change the view of the table. A user must specify the number of partitions in the vector attributes panel and then complete the correct number of partitions in the partition table. The same is true for eventcounts and sequencers. A similar issue exists with the number of processes field and the process field of the partition table. A user must add a single process and then ensure that at least one subject is filled out in the process window. Fortunately, when the vector is checked (either by the user clicking the check button or attempting to export the vector), the tool will notify the user of such errors and prompt the user to fix these errors.

C. FUTURE WORK

Although there will probably be multiple changes, enhancements, and refinements made to the design of the configuration vector tool in the future, this section focuses on the suggested next steps with regards to the interface, features, and documentation. Thus, this section is divided into four sections to address the interface, additional features, refinements, and documentation.

1. Interface

The current interface for the configuration vector tool is based on tables. As discussed in Section A of this chapter, the tables implemented for this prototype were constructed by taking advantage of the NetBeans GUI builder. Unfortunately, tables implemented using the NetBeans GUI builder are quite basic and lack advanced customization features. This hindered the developer's ability to finely control the display of individual tables. Thus, the first recommendation for the next version of the configuration vector tool is a complete rewrite of the code used to create the tables. Perhaps a new Java class should be created that creates tables. This would allow for maximum code reuse and allow future development of tables to be easier. With a customized tables class, the developer should have the ability to create advanced features such as hiding rows that are not specifically defined by the user of the configuration vector tool.

Another addition that should be added to the interface is a message panel below the tables. This panel would be read-only to the user of the configuration vector tool and display only informational messages. The messages displayed in this area of the interface would contain information specific to the field selected by the user. This would display the bounds (as necessary) for the specific field as well as a brief description of the selected field. This addition would increase the user's awareness of the data required in order to create a configuration vector.

The final interface enhancement that should be added is a *preferences window*. The *preferences window* should allow the user to specify default directories for exporting or saving configuration vector files (both binary and human-readable). In addition, the

preferences window should allow the user to set the fields that are read-only in the main interface. These items include the version and path to the LPSK kernel file (PL0 executable path of the subject table).

2. Additional Features

Many additional features could be added to the configuration vector tool. For the next version of the tool, the human-readable vector file generated by the tool should be converted to an XML format, preset configuration vectors should be added, and a visual representation of the vector created by the tool should be displayed to the user.

A vector file formatted in XML is an easy way to create a vector file that is both human-readable and machine-readable. Following an XML standard would allow a more streamlined approach to saving specific configurations for future use. The first step is creating an XML schema for a valid vector file. Then, the current *save/save as* function in the tool should be changed to save the file to the XML format. The *open* function should also be changed to only read in valid XML vector files.

The next additional feature is *presets*. Presets allow the user to create a valid configuration vector quickly and easily. Presets also enable the user to start from a known template and modify the data as necessary. There should be two types of presets implemented in the next version. The first is a set of default presets. These presets are those that ship with the configuration vector tool. The second type of presets are those that are defined by the user. Similar to opening a pre-existing configuration vector, user-defined presets allow the user to save custom presets for future use.

The final feature that should be added to the configuration vector tool is a visual representation of the current configuration created in the tool. In other words, the visual representation would display a graphical picture of the configuration vector the user has created. This feature should be invoked by default when the check button is pressed. When the export button is pressed, the user should be given the option of graphically displaying the configuration vector or proceeding directly to exporting the file. This would enable the user to actually "see" what has been created and to visually verify the configuration.

3. Refinements

Aside from the interface changes and the additional features previously discussed, the configuration tool also requires minor refinements in order to make it a more complete product. All of these items increase the usability of the tool. These refinements are listed below.

- Center column values in all tables
- Increase the text size of the tables
- Add color-coding to permissions window (e.g., NA colored red, RW colored green, etc.)
- Add color-coding to identify errors in table cells
- Add tooltips to all fields and buttons
- Tooltips for a partition should contain all relevant data for that partition
- Refine error messages
- Add the capability to create a message authentication code for an input configuration vector
- Refine selection behavior of the tables
- Refine resizing of the main window
- All sub-windows (i.e., process window and permissions window) must restrict focus and not allow a user to click on the main window without completing the current sub-window

4. Documentation

The configuration vector tool requires two types of documents: a configuration vector reference manual and a configuration vector tool user guide. The configuration vector reference manual should contain information specific to the configuration vector. This reference manual should contain the descriptions and bounds for all variables of the configuration vector. It should also provide several examples of valid configuration vectors and should describe how those vectors implement a particular policy. This will allow a trusted user to better understand exactly what he or she is trying to create. The configuration vector tool user guide should contain the instructions on how to correctly use the configuration vector tool to create or manipulate a configuration vector.

D. CONCLUSION

The prototype configuration vector definition tool was designed to meet the need for a better way to create the configuration vectors used to initialize the LPSK. Although the prototype needs more work before it should be considered operational, the initial design is complete. Before the prototype can be considered fully functional, four problems need to be addressed. First, the tables created using the NetBeans GUI builder need to be rewritten to allow the developer complete control of all cells in the table. Second, the human-readable configuration vector created by the tool should be converted to XML, which will provide more flexibility. Third, a visual representation the configuration allowing the user to visualize the configuration before exporting a binary configuration vector file is needed. Finally, to increase the usability of the tool, the refinements discussed should be implemented. Since the prototype follows the MVC design paradigm (see Chapter II, Section B), the backend of the tool is completely separate from the GUI front-side. Thus, modifications to the tool's GUI do not affect the underlying data structures. This allows for the development of GUI improvements as necessary.

This research developed the initial design of the LPSK configuration vector tool and created a partially functional prototype. The goal of creating a graphical interface for the configuration vector tool was achieved. However, the prototype must be refined before becoming operational. The prototype was designed to be robust enough to handle such changes without significant effort.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX

The following four pages contain the code for the `lpsk.h` file for the Least Privilege Separation Kernel (LPSK). All structures as well as all constants were referenced from this file to create the configuration vector tool.

```

1
2 #ifndef LPSK_H
3 #define LPSK_H
4
5 #define MAX_DESC      32      // Max length of description string
6 #define MAX_NAME      32      // Max length of exported object name
7 #define NUM_PLS       4       // Number PLS supported by CPU
8 #define MAX_DSEGS     64      // Max # of dsegs
9 #define MAX_MSEGS     32      // Max # of msecs
10 #define MAX_DEVICES   32      // Max # of devices
11 #define MAX_PATH      64      // Max path for file name
12 #define MAX_PARTITIONS 8      // Max partitions
13 #define MAX_PROCESSES 1       // Max processes per partition
14 #define MAX_STOP_MSG  1024    // Max length of halt message
15 // Max # of subjects
16 // One for each process, and each privilege level (except PL0)
17 #define MAX_SUBJECTS   (MAX_PARTITIONS * MAX_PROCESSES * (NUM_PLS-1))
18
19 /* LPSK synchronization constants */
20 #define MAX_EVENTCOUNTS 32
21 #define MAX_SEQUENCERS  32
22
23 /* LPSK signal constants */
24 #define MAX_SIGNALS      32
25 #define NUM_KERNEL_SIGNALS 2
26 /* The kernel signals. These can only be sent by the kernel
27  * The ordering here shows signal interrupt priority.
28  * The lowest priority is listed first, highest last.
29  */
30 #define LPSK_SAK_SIGNAL   (MAX_SIGNALS)
31 #define LPSK_KILL_SIGNAL  (MAX_SIGNALS + 1)
32
33 #define TRUE 1
34 #define FALSE 0
35 // #define NULL((void*)0)
36
37 // The configuration vector magic #
38 #define CONFIG_MAGIC 0xcfcfcfcf
39
40 // The following are permissions that can be assigned to data segments
41 // memory segments, subjects, eventcounts, and sequencers
42 #define NA 0      // No Access
43 #define RO 1      // Read only
44                // Read/Await eventcount
45 #define RW 2      // Read and Write
46                // Signal subject
47                // Read/Await/Advance eventcount
48                // Ticket sequencer
49 #define WO 3      // Signal subject
50                // Advance eventcount
51
52 #define MAJ_KEYBOARD 1      // Major device # for keyboards
53 #define MAJ_SCREEN 2       // Major device # for screen
54
55
56 // Call interface return values
57 #define NO_ERROR      0
58 #define LPSK_NO_WRITE 1
59 #define LPSK_NOT_MAPPED 2
60 #define LPSK_BAD_MAJOR 3
61 #define LPSK_BAD_MINOR 4

```

```

62 #define LPSK_NOT_RANDOM      5
63 #define LPSK_NO_DATA         6
64 #define LPSK_READ_FAILURE    7
65 #define LPSK_WRITE_FAILURE    8
66 #define LPSK_BAD_OFFSET      9
67 #define LPSK_DEVICE_END     10
68 #define LPSK_BAD_SIGNAL_ID   11
69 #define LPSK_BAD_ADDRESS     12
70 #define LPSK_BAD_POINTER     13
71 #define LPSK_BAD_MASK        14
72 #define LPSK_BAD_MASK_CNT    15
73 #define LPSK_NOT_REGISTERED  16
74 #define LPSK_BAD_SUBJECT_ID  17
75 #define LPSK_BAD_SUBJECT_CNT 18
76 #define LPSK_NOT_ALLOWED     19
77 #define LPSK_BAD_EVENTCOUNT 20
78 #define LPSK_BAD_SEQUENCER   21
79
80 typedef unsigned short int selector_type;
81 typedef unsigned int boolean;
82 typedef unsigned int major_type;
83 typedef unsigned int minor_type;
84
85 // The following is used to declare eventcount/sequencer
86 // requests in the configuration vector
87 //
88 typedef struct {
89     char          name[MAX_NAME];      // Name of object
90     unsigned int  pl;                  // privilege level of object
91     unsigned int  perms[MAX_PARTITIONS]; // permissions for each partition
92 } synchronization_struct;
93
94 // The following structure defines an exported object
95 //
96 typedef struct {
97     unsigned int  object_identifier;
98     char          object_name[MAX_NAME];
99     unsigned int  object_permission; // access permission for the object
100 } object_id_struct;
101
102 // The following is used to declare dseg requests in the configuration vector
103 //
104 typedef struct {
105     unsigned int partition;      // partition to load in
106     unsigned int identifier;     // identifier
107     unsigned int pl;             // privilege level to load in
108     char         path[MAX_PATH]; // location on disk
109     unsigned int perms[MAX_PARTITIONS]; // permissions for each partition
110 } datafile_struct;
111
112 // The following is used to declare mseg requests in the configuration vector
113 //
114 typedef struct {
115     unsigned int partition;      // partition to load in
116     unsigned int identifier;     // identifier
117     unsigned int pl;             // PL to allocate mseg in
118     unsigned int size;           // Size of requested mseg
119     unsigned int perms[MAX_PARTITIONS]; // access list for mseg
120 } memory_struct;
121
122 // The following structure is used to "define" a subject in the config vector

```



```

123 //
124 typedef struct {
125     char    exe_path[MAX_PATH];    // location of executable file
126     char    gate_path[MAX_PATH];  // location of gate information
127 } subject_struct;
128
129 // The following is used to "define" a process in the configuration vector
130 //
131 typedef struct {
132     unsigned int    identifier;        // Process identifier
133     unsigned int    num_subjects;      // # of subjects in the process
134     subject_struct  code[NUM_PLS];
135 } process_struct;
136
137 // The following structure is used to "define" a partition in config vector
138 //
139 typedef struct {
140     char    description[MAX_DESC];    // description of partition
141     unsigned int    identifier;        // Partition identifier
142     unsigned int    time_slice;        // Fixed scheduling slice
143     unsigned int    max_memory;        // max memory partition can use
144     boolean    active;                // TRUE=active, FALSE=passive
145     unsigned int    num_processes;     // # of processes in the partition
146     process_struct processes[MAX_PROCESSES]; // Processes definitions
147 } partition_struct;
148
149 // The following structure defines the permissions
150 // a subject has to various resources
151 //
152 typedef struct {
153     unsigned int    subj_perm[MAX_SUBJECTS];    // Other subjects
154     unsigned int    dseg_perm[MAX_DSEGS];       // data segments
155     unsigned int    mseg_perm[MAX_MSEGS];       // memory segments
156     unsigned int    evct_perm[MAX_EVENTCOUNTS]; // eventcounts
157     unsigned int    seq_perm[MAX_SEQUENCERS];    // sequencers
158 } subj_res_perm_struct;
159
160 // The following structure is an LPSK configuration vector
161 //
162 typedef struct {
163     unsigned int    version;                // The format version
164     unsigned int    magic;                  // The structure magic #
165     unsigned int    num_partitions;         // The # of partitions
166     int    tpa_partition;                   // The TPA partition
167     partition_struct partitions[MAX_PARTITIONS]; // All the partition defn's
168     datafile_struct datafile[MAX_DSEGS];     // additional data segments
169     memory_struct    memory[MAX_MSEGS];      // additional memory
170     unsigned int    num_eventcounts;         // The # of eventcounts
171     synchronization_struct eventcounts[MAX_EVENTCOUNTS]; // eventcount data
172     unsigned int    num_sequencers;          // The # of sequencers
173     synchronization_struct sequencers[MAX_SEQUENCERS]; // sequencer data
174     // Partition -> Partition permissions
175     unsigned int    part_perm[MAX_PARTITIONS][MAX_PARTITIONS];
176     // subject -> resource permissions
177     subj_res_perm_struct    subj_perm[MAX_SUBJECTS];
178 } vector_struct;
179
180
181 // The following structure defines a particular device
182 //
183 typedef struct {

```

```

184     major_type    major;
185     minor_type    minor;
186 } device_struct;
187
188 // The following structure defines a particular subject
189 //
190 typedef struct {
191     unsigned int process_identifier;    // Which process is the subject in
192     unsigned int pl;                  // privilege level of subject
193 } subject_id_struct;
194
195 // The following is used to pass dseg and mseg info to other privilege levels
196 //
197 typedef struct {
198     selector_type selector;            // Selector to a segment
199     unsigned int size;                // Size of the segment
200     unsigned int perms;               // Permissions to segment
201     char id[MAX_PATH];                // Identifier for the segment
202 } segment_struct;
203
204 // The following is used to pass outer-ring return info
205 // to other privilege levels
206 //
207 typedef struct {
208     selector_type code_sel;           // Code segment selector
209     unsigned int eip;                 // Initial IP value
210     selector_type stack_sel;          // Stack segment selector
211     unsigned int esp;                 // Initial SP value
212 } outer_ring_return_struct;
213
214 // The following is the structure passed to either PL1 or PL2
215 // by the LPSK
216 //
217 typedef struct {
218     unsigned int version;             // The format version
219     int tpa_partition;                // The TPA partition identifier
220     unsigned int process_id;          // The process identifier
221     unsigned int num_partitions;       // The # of partitions
222     unsigned int num_devices;         // The # of attached devices
223     device_struct devices[MAX_DEVICES]; // All the device defn's
224     segment_struct segs[MAX_DSEGS+MAX_MSEGS]; // additional segments
225     unsigned int num_subjects;
226     object_id_struct subjects[MAX_SUBJECTS];
227     unsigned int num_eventcounts;
228     object_id_struct eventcounts[MAX_EVENTCOUNTS];
229     unsigned int num_sequencers;
230     object_id_struct sequencers[MAX_SEQUENCERS];
231     outer_ring_return_struct orr[2];  // PL2, PL3 outer-ring return info
232 } outer_ring_struct;
233

```


THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] C. E. Irvine, T. E. Levin, T. D. Nguyen, and G. W. Dinolt, "The Trusted Computing Exemplar Project," in *Proceedings of the 5th IEEE Systems*, (Military Academy, West Point, NY), pp. 109-115, IEEE Computer Society Press, June 2004.
- [2] C. E. Irvine and K. Levitt, "Trusted Hardware: Can It Be Trustworthy?" in *Proceedings of the 44th Annual Design Automation Conference, DAC '07*, ACM, 2007.
- [3] Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance requirements, Version 2.1, CIMB-99-033, August 1999.
- [4] U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.03, Information Assurance Directorate, 2007.
- [5] S. Burbeck, "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)," 4 March 1997. Available: <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html> (accessed 8 October 2009).
- [6] Java BluePrints Model-View-Controller, Sun Microsystems, Inc., 2002. Available: <http://java.sun.com/blueprints/patterns/MVC-detailed.html> (accessed 15 October 2009).
- [7] A. Fowler, "A Swing Architecture Overview," Sun Microsystems, Inc., 2009. Available: <http://java.sun.com/products/jfc/tsc/articles/architecture/> (accessed 8 October 2009).
- [8] Microsoft.NET, Microsoft, 2009. Available: <http://www.microsoft.com/NET/> (accessed 3 August 2009).
- [9] Cocoa, Apple Inc., 2009. Available: <http://developer.apple.com/cocoa/> (accessed 3 August 2009).
- [10] wxPython. 22 May 2009. Available: <http://www.wxpython.org/> (accessed 2 August 2009).
- [11] Java SE Downloads, Sun Developer Network (SDN), Sun Microsystems, 2009. Available: <http://java.sun.com/javase/downloads/index.jsp> (accessed 3 October 2009).
- [12] What is wxPython?, wxPython, 2009. Available: <http://www.wxpython.org/what.php> (accessed 2 August 2009).

- [13] wxPyWiki. 25 Nov 2009. Available: <http://wiki.wxpython.org/> (accessed 2 August 2009).
- [14] SDN: A Community for Sun Developers, Sun Microsystems, 2009. Available: <http://developers.sun.com/> (accessed 18 October 2009).
- [15] wxGlade: a GUI builder for wxWidgets, 13 October 2009. Available: <http://wxglade.sourceforge.net/> (accessed 14 October 2009).
- [16] Boa Constructor, 2003. Available: <http://boa-creator.sourceforge.net/> (accessed 14 October 2009).
- [17] NetBeans IDE 6.7 Connects Developers, NetBeans, 2009. Available: <http://www.netbeans.org/index.html> (accessed 17 October 2009).
- [18] Introduction to Apple Human Interface Guidelines. Apple Inc., 20 August 2009. Available: <http://developer.apple.com/mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines/XHIGIntro/XHIGIntro.html> (accessed 10 August 2009).

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Susan Alexander
OASD/NII DOD/CIO
Washington, DC
4. George Bieber
OSD
Washington, DC
5. Kris Britton
National Security Agency
Fort Meade, Maryland
6. Ed Bryant
Unified Cross Domain Management Office
Maryland
7. John Campbell
National Security Agency
Fort Meade, Maryland
8. Deborah Cooper
DC Associates, LLC
Roslyn, Virginia
9. Grace Crowder
NSA
Fort Meade, Maryland
10. Louise Davidson
National Geospatial Agency
Bethesda, Maryland

11. Vincent J. DiMaria
National Security Agency
Fort Meade, Maryland
12. Rob Dobry
NSA
Fort Meade, Maryland
13. Jennifer Guild
SPAWAR
Charleston, South Carolina
14. CDR Scott Heller
SPAWAR
Charleston, South Carolina
15. Dr. Steven King
ODUSD
Washington, DC
16. Steve LaFountain
NSA
Fort Meade, Maryland
17. Dr. Greg Larson
IDA
Alexandria, Virginia
18. Dr. Carl Landwehr
National Science Foundation
Arlington, Virginia
19. Dr. John Monastra
Aerospace Corporation
Chantilly, Virginia
20. John Mildner
SPAWAR
Charleston, South Carolina
21. Dr. Victor Piotrowski
National Science Foundation
Arlington, Virginia

22. Jim Roberts
Central Intelligence Agency
Reston, Virginia
23. Ed Schneider
IDA
Alexandria, Virginia
24. Mark Schneider
NSA
Fort Meade, Maryland
25. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC
26. Ken Shotting
NSA
Fort Meade, Maryland
27. Dr. Ralph Wachter
ONR
Arlington, Virginia
28. John Santos
CERDEC S&TCD Information Assurance Division
Fort Monmouth, New Jersey
29. Ernie Brickell
Intel
Hillsboro, Oregon
30. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, California
31. Paul C. Clark
Naval Postgraduate School
Monterey, California
32. Terrence M. Welliver
SFS students: Civilian, Naval Postgraduate School
Monterey, California